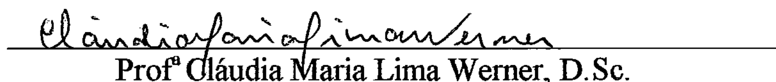
UMA ARQUITETURA DE APOIO PARA ANÁLISE DE MODELOS ORIENTADOS A OBJETOS

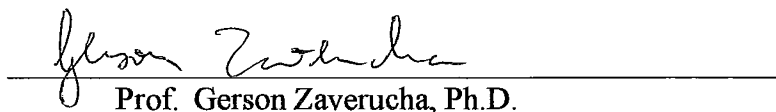
Alexandre Luis Correa

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

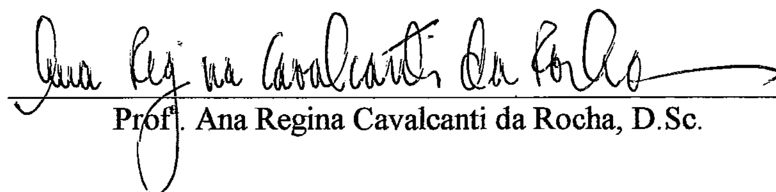
Aprovada por:



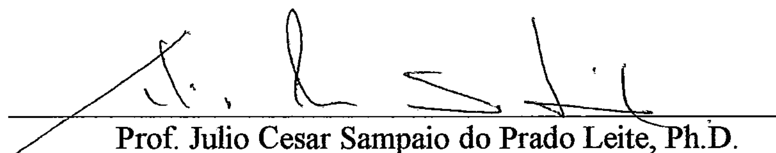
Profª Cláudia Maria Lima Werner, D.Sc.



Prof. Gerson Zaverucha, Ph.D.



Profª. Ana Regina Cavalcanti da Rocha, D.Sc.



Prof. Julio Cesar Sampaio do Prado Leite, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

JULHO DE 1999

CORREA, ALEXANDRE LUIS

Uma arquitetura de apoio para análise de modelos orientados a objetos

[Rio de Janeiro], 1999

VI, 133 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1999)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Orientação a Objetos
2. Padrões
3. Anti-Padrões
4. Projeto de Software Orientado a Objetos

I. COPPE/UFRJ II. Título (série)

Agradecimentos

À Professora Cláudia Werner e ao Professor Gerson Zaverucha, pela orientação baseada em confiança, respeito e permanente incentivo. Agradeço também pela amizade, paciência, dedicação e pelas lições que pude tirar desta convivência.

À Professora Ana Regina Rocha, pelo conhecimento transmitido direta e indiretamente neste período, e pela honra de ter sua presença na banca de defesa de tese.

Ao Professor Julio Leite, por honrar-me com sua presença na banca de defesa de tese.

Ao Professor Guilherme Travassos, pelos ensinamentos durante este período.

Aos meus pais Paulo e Dirce, ambos fontes inesgotáveis de determinação, esperança, otimismo e amor. Agradeço por compreenderem meu mau humor, especialmente nos últimos meses deste trabalho, e dedico a eles toda a minha formação.

À minha noiva Glória, pelo amor, carinho, apoio e compreensão nesta caminhada tão árdua.

Ao Professor Éber Schmitz pelo apoio, confiança e amizade.

Ao velho companheiro, Professor Ricardo Bianchini, que sempre insistiu e me incentivou a cumprir este desafio.

Aos amigos que fiz na COPPE durante este período, em especial, Alexandre Dantas, José Ricardo Xavier, Leonardo Murta, Márcio Barros, Regina Braga e Rodrigo Basílio.

A todos os professores, funcionários e colegas do Programa de Sistemas que, direta ou indiretamente, me deram suporte para a realização deste trabalho.

Aos amigos da MRS Logística S.A. e VARIG S.A. pela compreensão e apoio.

A todos os demais amigos que, de alguma forma, colaboraram para a conclusão dessa etapa.

À CAPES pelo apoio financeiro.

Agradeço finalmente a DEUS, por iluminar o meu caminho e por me dar a graça de desfrutar de saúde e paz.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA ARQUITETURA DE APOIO PARA ANÁLISE DE MODELOS ORIENTADOS A OBJETOS

Alexandre Luis Correa

Julho/1999

Orientadores: Cláudia Maria Lima Werner
Gerson Zaverucha

Programa: Engenharia de Sistemas e Computação

O paradigma da orientação a objetos passou a ocupar lugar de destaque no desenvolvimento de software nos anos 90. Embora dois dos grandes atrativos deste novo paradigma sejam o grande potencial de reutilização e a facilidade de manutenção do software, a realidade mostra que várias organizações não conseguiram atingir plenamente estes objetivos. Hoje, já é possível encontrar vários sistemas de difícil manutenção, apresentando estruturas rígidas e inflexíveis, mesmo tendo sido desenvolvidos com métodos e linguagens de programação orientadas a objetos.

Este trabalho apresenta um estudo sobre técnicas, não enfatizadas por métodos e ferramentas CASE de suporte ao desenvolvimento OO, que constituem importante fonte de conhecimento para os projetistas na elaboração de projetos mais flexíveis e reutilizáveis. É apresentada uma abordagem para a integração deste conhecimento a ferramentas CASE de projeto OO, possibilitando a detecção de construções padronizadas correspondentes a boas (padrões) e más (anti-padrões) soluções de projeto. A utilização prática desta abordagem é ilustrada por um protótipo da ferramenta correspondente à implementação desta abordagem, denominada OOPDTool, e um estudo de caso realizado em dois projetos OO.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A SUPPORT ARCHITECTURE FOR OBJECT ORIENTED MODELS ANALYSIS

Alexandre Luis Correa

July/1999

Advisors: Claudia Maria Lima Werner
Gerson Zaverucha
Department: Systems and Computer Engineering

The object oriented paradigm has become very popular on this decade. Although the two main promises of this paradigm are great reuse potential and ease of software maintenance, many organizations have not achieved satisfactory results. Today, it is possible to find many systems that are hard to maintain, presenting rigid and inflexible structures, despite the use of object oriented methods and languages.

This work describes a research developed to identify techniques, not emphasized by OO methods and CASE tools, that represent an important source of knowledge to OO designers. An approach to the integration of these techniques to OO CASE tools is presented. This approach allows the identification of good (patterns) and bad (anti-patterns) constructions present in an OO design. The practical use of the proposed approach is shown by a prototype corresponding to the implementation of this approach, called OOPDTool, and a case study involving two OO projects.

ÍNDICE

1. Introdução 1

2. Heurísticas, Padrões, Anti-Padrões de Projeto Orientado a Objetos 7

2.1. Introdução 7

2.2. Heurísticas de Projeto Orientado a Objetos 8

2.3. Padrões 12

2.4. Anti-Padrões 17

2.5. Relação entre Heurísticas, Padrões e Anti-Padrões 21

2.6. Ferramentas de Suporte Existentes 23

2.6.1. Apoio à Engenharia Reversa ou Reestruturação 24

2.6.1.1. Pat 25
2.6.1.2. KT 25
2.6.1.3. OMT-Tool 26
2.6.1.4. DEPARS: Design Patterns Recognition System 27
2.6.1.5. Design Pattern Tool (DPT) 28

2.6.2. Apoio ao Desenvolvimento com Padrões 29

2.6.2.1. Gerador automático de código a partir de padrões de projeto - IBM 29
2.6.2.2. Universidade de Waterloo – Ferramenta para definição e aplicação formal de padrões de projeto 31
2.6.2.3. Universidade de Utrecht – Ambiente de suporte ao desenvolvimento de sistemas baseado em padrões 33
2.6.2.4. Framework Studio 34

2.7. Conclusões 34

3. Uma Arquitetura de Apoio para a Análise de Modelos Orientados a Objetos ... 37

3.1. Introdução 37

3.2. Metamodelo 40

3.2.1. Pacotes 42
3.2.2. Classificadores 42
3.2.3. Visibilidade de elementos dos pacotes 44
3.2.4. Atributos 44
3.2.5. Operações 46
3.2.6. Herança e Realização 47
3.2.7. Dinâmica dos objetos 48

3.3. Extrator de Projeto 51

3.4. Gerador de Fatos 54

3.5. Captura de Conhecimento 55

3.6. Analisador 61

3.7. Conclusões 63

4. OOPDTool – Uma Ferramenta de Apoio à Detecção de Padrões e Anti-Padrões 65

4.1. Introdução 65

4.2. Interoperabilidade com a ferramenta CASE 66

4.3. OOPDTool – Visão Geral 68

4.4. OOPDTool – Captura de Conhecimento 72
4.4.1. Heurísticas 72
4.4.2. Padrões de Projeto 75
4.4.3. Anti-Padrões 79

4.5. OOPDTool – Gerador de Fatos 83

4.6. OOPDTool - Analisador 87

4.7. Estudo de caso 96
4.7.1. Visibilidade Pública de Atributo 97
4.7.2. Visibilidade de atributo para subclasses 97
4.7.3. Herança de classe concreta 98
4.7.4. Classe "Entidade" definida a partir de uma classe utilitária 99
4.7.5. Exposição da implementação de uma coleção 99
4.7.6. Superclasse dependente de Subclasse 100
4.7.7. Violação da Lei de Demeter 101
4.7.8. Objeto com escopo global 102
4.7.9. Factory method ao invés de Abstract Factory 103
4.7.10. Factory method ao invés de Prototype 104

4.8. Conclusões 106

5. Conclusões 108

REFERÊNCIAS BIBLIOGRÁFICAS 114

Apêndice I - METAMODELO – Representação em Prolog 123

ÍNDICE DE FIGURAS

Figura 1: Anti-padrão *"Blob"* 20
Figura 2: Estrutura de um projeto antes da aplicação do padrão *"Facade"* 21
Figura 3: Estrutura de um projeto após a aplicação do padrão *"Facade"* 22
Figura 4: Exemplo de transformação do DPT 28
Figura 5: Descrição formal de um padrão 32
Figura 6: Componentes da abordagem 39
Figura 7: Elementos básicos do metamodelo 41
Figura 8: Dependência circular entre pacotes. 42
Figura 9: Exemplo de atributo e pseudo-atributo 44
Figura 10: Relações entre classificadores 47
figura 11: Instanciação de um objeto 48
figura 12: Destruição de um objeto 49
Figura 13: Invocação de operação 49
Figura 14: Acesso a um atributo de um objeto 51
Figura 15: Exemplo de métodos e seus estereótipos 52
Figura 16: Exemplo de extração de informações sobre troca de mensagens. 53
Figura 17: Predicados utilizados para expressar fatos sobre um projeto OO 55
Figura 18: padrão *"Abstract Factory"* 58
Figura 19: Detecção de um participante no papel de *AbstractFactory* 58
Figura 20: Detecção de um participante no papel de *ConcreteFactory* descendente de um *AbstractFactory* 58
Figura 21: Detecção de um participante no papel de *AbstractProduct* 59
Figura 22: Detecção de um participante no papel de *ConcreteProduct*, descendente de um *AbstractProduct*. 59
Figura 23: Verifica se um classificador possui uma operação que retorna um *AbstractProduct*. 59
Figura 24: Verifica se um classificador possui uma operação que instancia um *ConcreteProduct*. 59
Figura 25: Identifica relações de herança direta ou indireta entre classificadores. 59
Figura 26: Anti-Padrão *"PublicVisibility"* 60
Figura 27: Anti-Padrão *"ProtectedVisibility"* 60
Figura 28: Anti-Padrão *"ExpositionOfAuxiliaryMethod"* 60
Figura 29: anti-padrão *"Global Scope Object"* 61
Figura 30: anti-padrão *"DisperseInstantiation"* 61
Figura 31: Aspectos de interoperabilidade do Rose98 67
Figura 32: Principais elementos da implementação do OOPDTool 69
Figura 33 – OOPDTool - modelo de classes 71
Figura 34: OOPDTool – menu principal 71
Figura 35: Catálogo de heurísticas 72
Figura 36: Janela de edição de uma heurística 73
Figura 37: Relacionamento entre heurísticas e anti-padrões 74
Figura 38: Relacionamento entre heurísticas e padrões de projeto 74
Figura 39: Catálogo de padrões 75
Figura 40: Janela de edição de padrão 76
Figura 41 : Associação de anti-padrões a um padrão 77
Figura 42: Parâmetros para a detecção de um padrão 77
Figura 43: Regra para detecção do padrão *"AbstractFactory"* 78
Figura 44: Associação entre padrões 79
Figura 45: Catálogo de anti-padrões 79
Figura 46: Janela de edição de anti-padrões 80
Figura 47: Anti-padrões - Soluções 81
Figura 48: Heurísticas associadas a um anti-padrão 81
Figura 49: Detecção de um anti-padrão 82
Figura 50: Extensão do Rose pela adição de propriedades 84
Figura 51 : Associação de um diagrama de colaboração a um método de uma classe 85
Figura 52: Diagrama de colaboração para uma operação 86

Figura 53: Extrato do diagrama de classes do editor de diagramas ... 86
Figura 54: Extrato da base de fatos gerada pelo OOPDTool ... 87
Figura 55: Seleção direta de anti-padrões ... 88
Figura 56: Seleção via heurísticas ... 89
Figura 57: Seleção via padrões ... 90
Figura 58: Resultado da pesquisa por um anti-padrão ... 91
Figura 59: Resultado da pesquisa por um padrão ... 92
Figura 60: Diagrama de classes – exemplo ... 92
Figura 61: Diagrama de colaboração – exemplo ... 93
Figura 62: Regra para detecção do padrão *Factory Method* ... 93
Figura 63: Extrato da Base de Fatos ... 94
Figura 64: Exemplo de instanciação da regra "*factoryMethodPattern*" ... 95
Figura 65: Visibilidade pública de atributos ... 97
Figura 66: Visibilidade de atributo para subclasses ... 98
Figura 67: Herança de classe concreta ... 98
Figura 68: Classe "Entidade" definida a partir de uma classe utilitária ... 99
Figura 69: Exposição da implementação de uma coleção ... 100
Figura 70: Superclasse dependente de Subclasse ... 100
Figura 71: Violação da lei de Demeter. ... 102
Figura 72: Objetos com escopo global ... 103
Figura 73: *Factory Method* ao invés de *Abstract Factory* ... 103
Figura 74: *Factory method* ao invés de *Prototype* ... 105
Figura 75: Exemplos de nomes qualificados ... 123

ÍNDICE DE TABELAS

Tabela 1: Quadro resumo do suporte a padrões, anti-padrões e heurísticas 36
Tabela 2: Quadro resumo atualizado 103

Capítulo 1

Introdução

Desenvolvimento de software é uma atividade que envolve riscos. Cerca de 1/3 dos projetos de software são cancelados, enquanto que apenas 1/6 dos projetos, aproximadamente, são bem sucedidos (JOHNSON, 1995). Vários projetos ultrapassam o prazo e/ou o orçamento em duas ou mais vezes o valor estimado. Os sistemas entregues freqüentemente não atendem a todos os requisitos ou, então, têm a sua manutenção muito difícil e cara. Como resultado, estes sistemas não conseguem acompanhar as constantes mudanças necessárias para os negócios (BROWN et al., 1998).

Neste contexto, a filosofia de reutilização desempenha um importante papel na busca por sistemas cuja produção e manutenção sejam realizadas de modo mais produtivo e fácil. Nós, seres humanos, aprendemos os benefícios da reutilização assim que começamos a enxergar o mundo de modo racional. Reutilizamos praticamente tudo que faz parte do nosso cotidiano: idéias, objetos, processos, etc. Um dos maiores indicadores deste fato é que sempre tentamos evitar o reinvento de algo que já exista, uma vez que isto faz pouco sentido, seja do ponto de vista econômico, intelectual ou pragmático. Entretanto, após cerca de 50 anos de evolução, a comunidade de software continua recriando o que já foi anteriormente criado (PRESSMAN, 1997).

A reutilização sistemática de software tem sido objeto de estudo e um dos objetivos da Engenharia de Software nos últimos anos. As disciplinas tradicionais de engenharia (Civil, Mecânica, Naval, entre outras) estão intrinsecamente baseadas em uma grande reutilização de elementos. O termo engenharia está relacionado com a criação de soluções eficientes, para problemas práticos, através da aplicação de conhecimento científico, construindo coisas a serviço da humanidade (PRESSMAN, 1997). Para que tal objetivo seja atingido, este conhecimento deve estar disponível de forma diretamente utilizável por um engenheiro, fornecendo respostas para questões comuns nas suas atividades práticas, permitindo a reutilização de conhecimento na

geração de soluções. Um projetista de automóvel, por exemplo, não projeta carros a partir das leis fundamentais da física, mas sim utilizando padrões de projeto bem sucedidos no passado.

Algumas condições são necessárias para que uma disciplina atinja a condição de engenharia, dentre as quais podemos destacar: a diferenciação entre essência e acidente, e o conhecimento de quais elementos são estáveis, isto é, quais elementos estão presentes nas várias soluções de problemas similares e quais destes elementos estáveis são importantes para a geração das soluções. A reutilização efetiva depende, portanto, do entendimento e representação dos pontos de pressão da estabilidade e variabilidade do domínio da aplicação (COPLIEN, 1996).

Neste processo de busca por soluções reutilizáveis e sistemas mais fáceis de manter, o paradigma de orientação a objetos passou a ocupar um papel de destaque. Este paradigma de desenvolvimento de software foi inicialmente proposto no final dos anos 60, levando vários anos para estar largamente difundido, e tornar-se, ao final da década de 90, o paradigma preferido de inúmeras organizações e profissionais. (PRESSMAN, 1997). Esta crescente popularidade do paradigma OO advém da promessa de podermos construir sistemas grandes e complexos, que sejam flexíveis não somente às constantes modificações de negócio, mas também à incorporação de evoluções tecnológicas. Esta promessa seria cumprida através do emprego de conceitos como abstração, encapsulamento, herança, polimorfismo, vinculação dinâmica, interfaces, entre outros, resultando em uma estrutura de software modular e reutilizável. Com isto, conseguiríamos uma dramática redução no esforço de manutenção e um significativo aumento de produtividade no desenvolvimento de novos sistemas, comparando-se com a utilização de paradigmas tradicionais de desenvolvimento (COAD e YOURDON 1991).

Os conceitos de orientação a objetos já são conhecidos pela comunidade acadêmica há algumas décadas, sendo descritos em inúmeras publicações disponíveis na literatura (JACOBSON et al., 1992), (BOOCH, 1994), (MEYER, 1997), (AMBLER, 1998), (RUMBAUGH et al., 1999). Durante a década de 90, surgiram diversos métodos de desenvolvimento de sistemas baseados nestes conceitos, dentre os quais podemos citar: *RDD* (WIRFS-BROCK et al., 1990), *OMT* (RUMBAUGH et al., 1991), *Coad/Yourdon* (COAD e YOURDON, 1991), *OOSE* (JACOBSON et al., 1992), *Shlaer*

e Mellor (SHLAER e MELLOR, 1992), *Booch* (BOOCH, 1994) e *Fusion* (COLEMAN et al., 1994). Uma descrição detalhada e um estudo comparativo de alguns destes métodos podem ser encontrados em (CLUNIE, 1997). Mais recentemente, foi criado o *RUP – Rational Unified Process* (JACOBSON et al., 1999), um processo de desenvolvimento de software resultado da unificação dos métodos de *Booch, OMT* e *OOSE,* incorporando, também, influências de vários outros. A notação utilizada por este processo para a geração dos modelos é a UML - *Unified Modeling Language* (OMG, 1998), também resultado de um esforço de unificação de dezenas de notações propostas pelos diversos métodos de desenvolvimento orientado a objetos.

Uma observação que pode ser feita a respeito destes diversos métodos é a ênfase dada aos elementos de representação dos conceitos de orientação a objetos, como por exemplo, os diversos tipos de diagramas, os elementos que compõem cada diagrama, o significado semântico de cada um destes elementos, o que eles podem representar, enfim, toda a informação necessária para a construção de um modelo correto de acordo com as construções da notação utilizada, como a UML, por exemplo. Entretanto, estes métodos oferecem pouca ajuda para os projetistas na árdua tarefa de gerar um modelo de objetos que dê origem a componentes flexíveis, reutilizáveis, fáceis de manter e de acomodar novas funcionalidades. Isto continua ocorrendo nos métodos mais recentes, como o RUP, por exemplo, que apenas define a necessidade de elaboração de um artefato denominado *"Design Guidelines"*, onde estariam instruções para o projetista no sentido de auxiliá-lo a construir um modelo OO de qualidade, sem definir, no entanto, quais são exatamente estas instruções. Uma exceção é a proposta de (D'SOUZA e WILLS, 1999), denominada *Catalysis* onde podemos encontrar uma ênfase maior em como atingir um projeto orientado a objetos flexível e reutilizável, com várias soluções catalogadas sob a forma de padrões.

Estes métodos, além de darem ênfase aos aspectos de representação dos elementos em diversos tipos de diagrama, fornecem diretrizes muito genéricas para o projetista. Um exemplo que pode ser encontrado em vários métodos é a diretriz *"Construa um sistema que espelhe o mundo real e mantenha-o assim"*. Esta diretriz é justificada pela potencial vantagem obtida através do paradigma orientado a objetos que consiste na redução da distância conceitual entre o mundo real e o computacional (WIRFS-BROCK et al., 1990).

Esta visão de mapeamento natural e direto do mundo real em um modelo de objetos é bastante útil quando estamos tentando entender o problema, normalmente em uma abordagem de análise. Entretanto, trabalhos como (GAMMA et al., 1995) e (MARTIN, 1995), apontam para o perigo de um projeto orientado a objetos baseado estritamente nestes elementos capturados diretamente do mundo real. O resultado pode ser um conjunto de elementos que implementem muito bem o mundo como ele é hoje, podendo ser, entretanto, pouco flexível a possíveis modificações futuras. As abstrações que surgem durante o projeto são a chave para a geração de componentes de software flexíveis e reutilizáveis (GAMMA et al., 1995).

Junto com estes métodos, várias ferramentas de suporte tornaram-se comercialmente disponíveis como o Rational Rose (RATIONAL, 1998) e Paradigm Plus (PLATINUM, 1999), por exemplo. Assim como os métodos, a ênfase destas ferramentas têm sido em apoiar a construção de um modelo correto do ponto de vista das construções semânticas disponíveis em várias linguagens de modelagem de software orientado a objetos como a UML, por exemplo. Entretanto, um modelo semanticamente correto não implica necessariamente em um modelo flexível e reutilizável e, em função deste fato, muitas organizações hoje dependem de revisões realizadas por especialistas para melhorar a qualidade do projeto, e evitar um esforço muito grande de manutenção futura. Estas revisões são caras e a existência de poucos especialistas disponíveis no mercado agrava ainda mais o quadro.

Desde o final da década de 80, muitos sistemas foram desenvolvidos utilizando estes métodos e ferramentas, sendo codificados em linguagens como, por exemplo, C++, Smalltalk e Java. Embora dois dos grandes atrativos deste novo paradigma sejam o grande potencial de reutilização e a facilidade de manutenção do software, a realidade é que boa parte destes sistemas não conseguiram atingir plenamente estes objetivos. Hoje, já é possível encontrar vários sistemas de difícil manutenção, apesar de terem sido desenvolvidos com técnicas e linguagens de programação orientadas a objetos (SCHULZ, 1998). Estes sistemas apresentam, ao contrário do esperado pela utilização deste paradigma, estruturas rígidas e inflexíveis, tornando difícil a incorporação de novas funcionalidades resultantes das inevitáveis modificações nos requisitos. Como exemplo, podemos citar alguns sistemas desenvolvidos por empresas como Nokia e Daimler-Benz. Estes sistemas são, hoje, objeto de estudo no contexto do projeto FAMOOS (SCG, 1998), que visa transformar sistemas legados orientados a objetos em

"*frameworks*" orientados a objetos.

Este quadro reforça a idéia de que projetar um software orientado a objetos flexível e reutilizável é uma tarefa difícil, exigindo uma gama de conhecimentos não enfatizados pelos métodos e ferramentas de suporte (MARTIN, 1995), (GAMMA et al., 1995). Muitos dos sistemas orientados a objetos problemáticos hoje existentes foram desenvolvidos por equipes, onde a maioria dos profissionais não possuía experiência suficiente para definir a construção mais adequada para um determinado problema, de forma a tornar o projeto mais flexível e robusto às modificações.

Esta tese apresenta um trabalho de pesquisa voltado à integração deste conhecimento, tão importante para a produção de sistemas orientados a objetos mais flexíveis e reutilizáveis, com métodos e ferramentas de suporte ao desenvolvimento OO. Este trabalho tem os seguintes objetivos:

- realizar um estudo sobre o conhecimento disponível na literatura que possa contribuir para a geração de projetos orientados a objetos mais flexíveis e reutilizáveis, além de investigar a existência de ferramentas que forneçam suporte para a utilização deste conhecimento.

- elaborar uma abordagem para a integração deste conhecimento às ferramentas CASE de projeto orientado a objetos, e construir um protótipo correspondente à implementação desta abordagem O objetivo é disponibilizar este conhecimento aos projetistas de forma organizada e interligada ao seu ambiente de desenvolvimento. Além disso, deseja-se também utilizar este conhecimento para auxiliar não somente o processo de restruturação de sistemas orientados a objetos problemáticos, mas também o desenvolvimento de novos sistemas. Este auxílio tem como objetivo a detecção de boas e más construções de projeto OO, isto é, construções correspondentes a soluções padronizadas para problemas recorrentes de projeto ou, por outro lado, construções que possam resultar em futuros problemas de manutenção e reutilização. Com isso, desejamos poder identificar pontos em um sistema que necessitem ser modificados de modo a torná-lo mais flexível e reutilizável, assim como facilitar o entendimento de sistemas grandes e mal documentados, através da detecção de vários padrões de construção utilizados no seu projeto.

- realizar um experimento que permita a investigação prática do impacto da utilização desta abordagem.

Esta tese está organizada em cinco capítulos e um apêndice. O primeiro deles definiu o objetivo do trabalho, suas motivações e organização.

O capítulo 2 – ***Heurísticas, Padrões, Anti-Padrões de Projeto Orientado a Objetos*** - apresenta os principais aspectos referentes aos conceitos de padrões, anti-padrões e heurísticas no contexto de projetos orientados a objetos. A importância e o impacto destes conceitos na produção de projetos mais flexíveis são discutidos, assim como os relacionamentos existentes entre estes conceitos. Por fim, é apresentado um estudo sobre algumas ferramentas que dão suporte à utilização destes conceitos na elaboração de projetos orientados a objetos.

O capítulo 3 – ***Uma arquitetura de apoio para a análise de modelos orientados a objetos*** - apresenta uma proposta para a integração de padrões, anti-padrões e heurísticas às ferramentas CASE de projeto orientado a objetos. Esta proposta consiste em uma arquitetura que visa disponibilizar para o projetista este conjunto de informações de forma estruturada, organizada e interligada com a ferramenta CASE de projeto, além de permitir a utilização deste conhecimento na detecção automática de padrões e anti-padrões em modelos de projeto elaborados com esta ferramenta CASE.

O capítulo 4 – ***OOPDTool: uma ferramenta de suporte à detecção de padrões e anti-padrões***, descreve a implementação da proposta descrita no capítulo 3, assim como um estudo de caso realizado em dois projetos orientados a objetos, visando uma investigação prática da utilidade da abordagem proposta.

O capítulo 5 – ***Conclusões***, apresenta as conclusões do trabalho, destacando-se suas contribuições, limitações e os trabalhos futuros que podem ser desenvolvidos.

O apêndice I – ***Metamodelo: Representação em Prolog*** descreve os predicados utilizados para a representação de elementos de um projeto orientado a objetos em Prolog. Esta representação é utilizada na implementação da ferramenta OOPDTool.

Capítulo 2

Heurísticas, Padrões, Anti-Padrões de Projeto Orientado a Objetos

2.1. Introdução

A construção de um projeto orientado a objetos é uma atividade complexa, se comparada à elaboração de um projeto estruturado convencional. Esta complexidade advém da variedade de objetos e das classes que compõem um projeto orientado a objetos, assim como da ampla gama de relacionamentos que podem existir entre eles. Além disso, cada tipo diferente de objeto, classe ou relacionamento possui sua própria semântica. Esta diversidade de elementos de projeto confere um poder de expressão que, se utilizado de forma apropriada, resulta em projetos robustos e com grande potencial de reutilização. Por outro lado, esta diversidade contribui para uma maior dificuldade no aprendizado e domínio da atividade de projeto orientado a objetos. (MARTIN, 1995)

Segundo GAMMA et al. (1995), projetar um software orientado a objetos é uma tarefa difícil, e projetar um software orientado a objetos reutilizável e flexível é ainda mais difícil. O projetista deve descobrir os objetos pertinentes, fatorá-los em classes em um nível de granularidade correto, definir as interfaces das classes, organizá-las em hierarquias e definir a dinâmica de colaboração entre os objetos, de forma que seja possível resolver o problema específico, mas de uma maneira suficientemente genérica para incorporar futuros problemas e requisitos. É desejável, também, evitar, ou pelo menos minimizar, a necessidade de reestruturação do projeto como resultado da incorporação de novas funcionalidades.

Embora seja difícil criar um projeto flexível e reutilizável na primeira tentativa, projetistas experientes conseguem obter resultados melhores do que os obtidos por iniciantes. A diversidade de elementos citada anteriormente faz com que os iniciantes se deparem com inúmeras alternativas possíveis e apresentem uma tendência para a

aplicação de técnicas não orientadas a objetos, em função não somente da maior familiaridade com estas técnicas, mas também da pressão exercida por um cronograma geralmente apertado, que não tolera "perdas de tempo" com a descoberta da melhor solução para um problema.

Este capítulo apresenta conceitos normalmente não enfatizados pelos métodos de desenvolvimento orientado a objetos, mas que representam importante fonte de conhecimento na produção de bons projetos. A seção 2.2 apresenta o conceito de heurísticas de projeto OO, ilustrando alguns exemplos encontrados na literatura. A seção 2.3 apresenta o conceito de padrões e sua aplicação no desenvolvimento de software. A seção 2.4 descreve o conceito de anti-padrões. A seção 2.5 discute como os conceitos de heurísticas, padrões e anti-padrões estão relacionados. Na seção 2.6, são apresentadas algumas ferramentas de suporte a utilização destes conceitos, em especial, de padrões. Finalizando este capítulo, a seção 2.7 apresenta algumas conclusões sobre a utilização destes conceitos no desenvolvimento de projetos orientados a objetos.

2.2. Heurísticas de Projeto Orientado a Objetos

Embora haja um consenso de que um método de desenvolvimento deva abordar diretrizes concretas de modo que os desenvolvedores possam produzir um bom projeto (HENDERSON-SELLERS, 1996), a grande maioria dos métodos e ferramentas CASE de apoio ao desenvolvimento orientado a objetos carecem deste suporte. Um problema típico de um projetista de software orientado a objetos ocorre quando ele, independente do método utilizado, deseja saber se o seu projeto está bom ou ruim do ponto de vista de flexibilidade em relação a futuras modificações e reutilização (RIEL, 1996). Como conseqüência desta falta de suporte, várias organizações acabam dependendo de revisões realizadas por especialistas, com o objetivo de detectar possíveis construções de projeto que possam gerar conseqüências negativas em termos de manutenção e reutilização futura (SCHULZ, 1998).

Quando um especialista analisa um determinado projeto, ele, de forma consciente ou inconsciente, investiga as construções do projeto em questão baseado em uma lista de heurísticas formuladas ao longo de sua experiência. Uma heurística de projeto é uma

diretriz resultante de conhecimento e experiência, que serve como uma espécie de conselho para tomada de decisões de projeto, sendo, portanto, de grande importância no sentido de guiar o projetista na elaboração de boas soluções ao longo do desenvolvimento (RIEL, 1996).

Em trabalhos como (JOHNSON e FOOTE, 1988), (MARTIN, 1995), (LIEBERHERR, 1996), (RIEL, 1996) e (MEYER, 1997), podemos encontrar um conjunto de heurísticas para a obtenção de um projeto orientado a objetos de qualidade. Uma heurística de projeto pode ser considerada como uma espécie de guia para decisões de projeto, descrevendo uma família de problemas em potencial e fornecendo dicas que auxiliam os projetistas a evitá-los. Heurísticas tais como "*Todos os dados devem estar escondidos dentro de sua classe*" (RIEL, 1996), (MARTIN, 1996a), norteiam as decisões de um projetista em direção a um projeto orientado a objetos mais flexível e reutilizável. É importante notar que uma heurística não deve ser vista como uma regra inviolável que deva ser seguida em todas as circunstâncias. Na verdade, possíveis violações a estas heurísticas devem ser consideradas como avisos ou indicadores de que alguma decisão de projeto pode ter sido incorretamente tomada.

MEYER (1997) define algumas heurísticas gerais associadas à modularização de projetos, dentre as quais destacam-se:

- *Todo módulo*[1] *deve se comunicar com o mínimo possível de outros módulos.* Considerando que Meyer faz uma correspondência direta entre módulos e classes, esta heurística visa minimizar o acoplamento entre as classes de um projeto orientado a objetos.

- *Dois módulos que se comunicam devem trocar o mínimo possível de informações.* Esta heurística visa minimizar a quantidade de informações trocadas entre pares de classes, como, por exemplo, os argumentos passados na invocação de uma operação de uma classe.

- *Se dois módulos se comunicam, este fato deve estar óbvio através da definição de A, B ou de ambos.* Esta heurística está relacionada à filosofia de "*projeto por contrato*", segundo a qual, os componentes de um sistema devem ser

[1] A palavra módulo neste contexto corresponde a uma classe ou componente de um projeto orientado a objetos (MEYER, 1997).

projetados de modo a cooperarem com base em contratos definidos de forma precisa (MEYER, 1997).

- *Toda informação a respeito de um módulo deve ser privativa do mesmo.* A importância desta heurística é a localização e impacto de modificações, ou seja, se tudo que diga respeito à implementação de um módulo não estiver visível para os seus clientes, qualquer modificação na implementação poderá ser realizada sem afetar os clientes do módulo.

- *Todo módulo deve ser fechado a modificações e aberto a extensões.* Segundo esta heurística, um módulo deve ser capaz de oferecer novas funcionalidades, sem que isto implique na modificação de outros módulos. Um software onde uma modificação resulte em uma cadeia de modificações em módulos dependentes revela uma estrutura frágil, rígida e não reutilizável (MARTIN, 1996a).

MARTIN (1995, 1996b, 1996c) define um conjunto de heurísticas para projetos orientados a objetos, dentre as quais podemos citar:

- *Inversão de Dependências*: módulos de alto nível não devem depender de módulos de mais baixo nível. Ambos devem depender de abstrações, sendo que os detalhes devem depender das abstrações, e não o contrário. No contexto de projeto orientado a objetos, esta heurística sugere que as classes concretas (implementações ou detalhes) dependam, sempre que possível, de abstrações (interfaces ou classes abstratas), e não de outras classes concretas (MARTIN, 1996b).

- *Clientes não devem ser forçados a depender de interfaces que eles não utilizam.* Se um cliente depende de interfaces que ele não utiliza, ele passa a ficar vulnerável a possíveis modificações nestas interfaces, o que resulta em um acoplamento indesejável entre dois módulos (MARTIN, 1996c).

- *As dependências entre pacotes[2] em um projeto devem ser na direção da estabilidade dos pacotes. Um pacote deve depender apenas de outros mais*

[2] Um pacote em um projeto orientado a objetos é um mecanismo geral utilizado para agrupar classes semanticamente próximas em um modelo orientado a objetos (BOOCH et al, 1999).

estáveis. A estabilidade de um pacote é dada pelo grau de dependência dos seus elementos em relação a elementos de outros pacotes e vice-versa. Quanto maior o número de elementos em um pacote que dependam de elementos de outros pacotes, maior a instabilidade do pacote (MARTIN, 1995).

- *Os pacotes mais estáveis devem ser os mais abstratos. Os pacotes mais instáveis devem ser os concretos.* Esta heurística está relacionada com a anterior, no sentido de que um pacote estável deve ser abstrato de modo que a sua estabilidade não impeça a sua extensão. Por outro lado, os pacotes mais instáveis devem ser compostos de classes concretas, uma vez que esta instabilidade permite que suas classes possam ser modificadas sem causar grande impacto no restante do sistema (MARTIN, 1995).

RIEL (1996) cataloga cerca de 60 heurísticas para projetos orientados a objetos distribuídas em assuntos:

a) Heurísticas gerais para classes e objetos, tais como:

- *Todos os dados devem ficar escondidos dentro de sua classe.* Esta heurística corresponde ao princípio de ocultação da informação (*"information hiding"*) descrito em vários trabalhos na literatura como, por exemplo, em (PARNAS, 1979) e (BOOCH, 1994).

- *Não coloque detalhes de implementação na área pública da classe.* Um exemplo de violação desta heurística é a exposição de métodos de implementação resultantes da fatoração da implementação de operações públicas da classe.

b) Heurísticas relativas à topologia de uma aplicação orientada a objetos, tais como:

- *Não crie classes/objetos com grande responsabilidade (classe "deus"), ou seja, classes que centralizam processamento ou dados.* De um modo geral, a existência de classes deste tipo indica uma distribuição não uniforme das responsabilidades do sistema.

- *Evite classes que contenham, na área pública, apenas métodos de acesso aos seus atributos.* Isto pode ser um indício de que dados e

comportamento não estejam sendo mantidos em um único lugar.

c) Heurísticas ligadas aos relacionamentos entre classes e objetos, tais como:

- *Uma classe deve conhecer os seus componentes, mas nunca quem a contém.* Ou seja, em uma relação todo/partes, o todo deve conhecer suas partes, mas uma parte não deve conhecer a classe correspondente ao todo.
- *Minimize o número de classes colaboradoras de uma determinada classe.*

d) Heurísticas ligadas à herança, tais como:

- *Todas as classes base em uma hierarquia de classes devem ser abstratas.*
- *Classes derivadas devem conhecer as suas classes base, por definição, mas classes base não devem ter qualquer tipo de conhecimento a respeito de suas classes derivadas.*

Estas heurísticas resumem, então, um importante conhecimento acumulado por seus proponentes, sendo de grande utilidade no sentido de auxiliar o projetista a tomar melhores decisões de projeto e evitar construções que possam trazer más conseqüências futuras. Entretanto, as centenas de heurísticas hoje existentes estão dispersas em várias publicações, descritas de maneira não uniforme e muitas vezes implícita, tornando difícil o seu reconhecimento como tal. Seria altamente desejável que o projetista tivesse estas heurísticas à sua disposição, de forma organizada, padronizada e integrada ao seu ambiente de desenvolvimento, e que, além disso, pudesse ser auxiliado na detecção de possíveis violações a estas heurísticas.

2.3. Padrões

Além das heurísticas de projeto orientado a objetos, a filosofia de padrões ou "*patterns*" (GAMMA et al., 1995) emergiu como uma possibilidade de se capturar o conhecimento de projetistas de software experientes, de forma a tornar mais fácil a reutilização de soluções bem sucedidas em problemas recorrentes de projeto de software.

O uso atual do termo padrão no contexto de software tem suas origens nos

trabalhos do arquiteto Christopher Alexander nas atividades de planejamento urbano e arquitetura de prédios (ALEXANDER 1977), (ALEXANDER 1979). Na definição original de Alexander, um padrão é "uma solução para um problema em um contexto". Entretanto, existem hoje muitas definições para o termo *"padrão"* na área de software. Sintetizando as definições de vários autores – (LEA, 1994), (GAMMA et al., 1995), (BUSCHMANN et al., 1996), (COPLIEN, 1996), (SCHMIDT et al., 1996), (VLISSIDES, 1997), podemos dizer que um padrão resolve um *problema recorrente*, em um determinado contexto, fornecendo uma *solução* que comprovadamente funcione (não especulações ou teorias), além de informar as *conseqüências*, ou seja, os resultados e compromissos da sua aplicação, e subsídios para que seja possível adaptar esta solução a uma variante do problema. Todo padrão deve ser identificado por um *nome*, possibilitando a formação de um vocabulário comum entre os seus usuários.

SHAW (1989) identifica um ciclo para o desenvolvimento de abstrações individuais que ajuda a entender onde a técnica de padrões atua no contexto de Engenharia de Software:

"Primeiro, problemas são resolvidos de forma ad hoc. Com o acúmulo de experiência, algumas soluções funcionam melhor que outras, e uma espécie de folclore é transmitido informalmente de pessoa para pessoa. Eventualmente, soluções úteis são entendidas de forma mais sistemática sendo, então, codificadas e analisadas. Isto permite o desenvolvimento de modelos que suportem a implementação automática destas soluções, e de teorias que possibilitem a extensão e generalização destas soluções. Por sua vez, isto permite um nível mais sofisticado da prática, o que possibilita a abordagem de problemas mais complexos. Estes problemas mais complexos são, então, novamente abordados de forma ad hoc, reiniciando o ciclo".

Um ponto chave para maximizar a reutilização e minimizar o esforço de manutenção de software, reside em projetá-lo tentando antecipar novos requisitos, assim como possíveis mudanças nos requisitos existentes. A utilização de padrões evita que estas modificações impliquem em grandes modificações na estrutura do software, uma vez que eles permitem que um aspecto do projeto do sistema possa variar de forma independente de outros, tornando o projeto mais robusto a um tipo particular de modificação. GAMMA et al. (1995), enumeram algumas causas comuns de inflexibilidade de projeto, apresentando os padrões que podem ser aplicados em tais

situações.

Como exemplo de uma causa comum de inflexibilidade de projeto, podemos citar a instanciação de um objeto através da especificação explícita de sua classe. Isto faz com que um determinado projeto fique dependente de uma implementação particular e não de uma interface. D'SOUZA e WILLS (1999) afirmam que, idealmente, cada classe deve depender apenas de interfaces e não de classes específicas, pois desta maneira, ela pode ser utilizada em conjunto com qualquer elemento que implemente as interfaces das quais ela depende, conferindo maior flexibilidade ao projeto. Entretanto, em uma aplicação orientada a objetos, as instâncias são criadas a partir de uma definição de classe, ou seja, quando desejamos criar um objeto, devemos dizer a qual classe ele pertence. Para manter a filosofia de dependência restrita a interfaces, este instante específico de dependência em relação a uma classe e, portanto, a uma implementação, pode ser encapsulado por padrões de projeto específicos para tal fim. Os padrões de projeto *"Abstract Factory"*, *"Factory Method"* e *"Prototype"* (GAMMA et al., 1995) são exemplos de padrões que podem ser aplicados no sentido de encapsular o processo de instanciação dos objetos, fazendo com que os clientes, ao invés de instanciarem diretamente os objetos, utilizem uma interface abstrata para tal.

Ao contrário das heurísticas, os padrões disponíveis hoje na literatura estão descritos de forma explícita e organizada. Embora existam várias formas de descrição de um padrão, tais como a forma *GoF* (GAMMA et al., 1995), a forma *POSA* (BUSCHMANN et al., 1996), e a forma *Alexander* (ALEXANDER, 1979), de um modo geral, toda descrição de um padrão deve conter as seguintes informações:

- **Nome:** todo padrão deve ser identificado por um nome. O nome tem fundamental importância, uma vez que ele serve como veículo de comunicação entre os projetistas, resumindo o conhecimento e a estrutura que estão por trás dele. Alguns padrões podem ser conhecidos por mais de um nome, ou seja, por sinônimos.

- **Problema:** descrição do problema ou questão específica que deve ser resolvida, e dos objetivos a serem atingidos, dado um determinado contexto. Descreve as forças e restrições relevantes, como elas interagem entre si e com os objetivos desejados.

- **Contexto:** descreve as condições nas quais o problema e sua solução se repetem, ou seja, define as situações onde um padrão é aplicável.

- **Solução:** descreve a solução do problema. Um padrão de projeto deve descrever a estrutura de solução em termos de seus participantes e da dinâmica de colaboração entre eles. Variantes ou especializações desta solução também são descritas nesta seção.

- **Conseqüências:** descreve os resultados obtidos com a aplicação do padrão, indicando como os seus objetivos são atingidos.

- **Padrões Relacionados:** referência para padrões que resolvam problemas similares e para padrões que possam auxiliar no refinamento de um determinado padrão. Por exemplo, o padrão *"Abstract Factory"* normalmente é implementado através da aplicação de dois outros padrões: *"Factory Method"* e *"Singleton"* (GAMMA et al., 1995).

- **Uso Conhecido**: exemplos da utilização deste padrão em sistemas.

Um conjunto de padrões forma um vocabulário que permite o entendimento e a comunicação de idéias. Em (BUSCHMANN et al., 1996), encontramos uma classificação para diferentes tipos de coleções de padrões, cada qual possuindo diferentes níveis de estrutura e interação: *catálogo de padrões, sistema de padrões, linguagem de padrões.*

Um catálogo de padrões corresponde a uma coleção plana de padrões, não revelando as interdependências e relacionamentos existentes entre eles. Um sistema de padrões adiciona a um catálogo de padrões uniformidade de descrição e uma estrutura mais aprofundada, expondo as interações existentes entre os padrões. Tanto sistemas de padrões como linguagens de padrões formam conjuntos coerentes de padrões fortemente relacionados para descrever e solucionar problemas em um domínio particular. Entretanto, uma linguagem de padrões torna um sistema de padrões mais robusto, fácil de compreender e completo. Um sistema de padrões pode focar em um tópico muito ou pouco abrangente, mas não precisa necessariamente possuir uma missão clara, o que pode resultar em uma maior dificuldade para encontrar relacionamentos entre os padrões, ou ainda, pode deixar várias lacunas não preenchidas no espaço do problema e,

portanto, não conter a solução completa. Uma linguagem de padrões, por sua vez, implica que seus padrões cubram todos os aspectos de importância em um domínio particular, devendo ser computacionalmente completa, fornecendo ao menos um padrão para todo aspecto de construção e implementação de um software no domínio coberto pela linguagem.

Em função da grande repercussão do livro sobre padrões de projeto (GAMMA et al., 1995), o foco principal da área de padrões dentro da comunidade de software tem sido neste tipo de padrão. Entretanto, existem padrões catalogados em diversos aspectos da Engenharia de Software como, por exemplo, organização do desenvolvimento, processos de desenvolvimento, planejamento de projeto, engenharia de requisitos, gerência de configuração, análise orientada a objetos, entre outros. Segundo BUSCHMANN et al. (1996), quanto maior o número de padrões em um sistema de padrões, maior é a dificuldade de entendê-los e utilizá-los. Neste contexto, um esquema de organização que permita agrupar padrões é importante para minimizar o esforço dos desenvolvedores em encontrar os padrões mais adequados para um determinado problema.

Um esquema de organização deve ser simples e fácil de aprender, ser constituído de apenas um pequeno número de critérios de classificação, fornecer um caminho que possibilite ao usuário encontrar um conjunto de padrões potencialmente aplicáveis em uma determinada situação, além de ser aberto à integração de novos padrões, o que eventualmente pode resultar em uma reestruturação da classificação já existente.

O esquema de organização proposto por BUSCHMANN et al. (1996) é composto por dois critérios de classificação: Categorias de padrões e Categorias de problemas. Em relação ao primeiro critério, são definidas três categorias baseadas nos níveis de abstração e detalhamento de suas construções:

- *Padrões arquiteturais* – expressam uma organização ou esquema estrutural para sistemas de software. Correspondem a estratégias de alto nível que afetam a estrutura e organização geral de todo o sistema

- *Padrões de projeto* – fornecem um esquema para refinar os subsistemas ou componentes de um sistema ou os relacionamentos entre eles. São aplicados para refinar e estender a arquitetura fundamental do sistema.

- *Idiomas* – transformam um projeto de software em um programa, escrito em uma determinada linguagem de programação. Correspondem a técnicas específicas de uma linguagem e do paradigma de programação que preenchem os detalhes de baixo nível da estrutura e comportamento dos componentes.

Na classificação dos padrões baseada em problemas, cada categoria corresponde diretamente a situações concretas de projeto, como, por exemplo, sistemas interativos (interação homem - máquina), sistemas distribuídos (infra-estrutura para sistemas em diferentes máquinas e locais), controle de acesso (padrões para controlar o acesso a serviços ou componentes).

Este esquema não é o único definido para a organização dos padrões. GAMMA et al. (1995) organizam os padrões em um esquema bidimensional segundo seu propósito (criação de objetos, estrutura ou comportamento) e seu escopo de aplicação (objetos ou classes). ZIMMER (1994) baseia sua classificação nos relacionamentos entre os padrões, como por exemplo, se um padrão A utiliza um padrão B ou se um padrão A é similar a um padrão B.

Assim como as heurísticas, os padrões também representam um importante conhecimento de grande utilidade no sentido de auxiliar o projetista a tomar melhores decisões de projeto e evitar construções que possam trazer más conseqüências futuras. Existem várias fontes de padrões disponíveis na literatura, como, por exemplo, (COPLIEN e SCHMIDT, 1995), (GAMMA et al. 1995), (BUSCHMANN et al., 1996), (VLISSIDES et al., 1996) e (MARTIN et al., 1997). Seria altamente desejável que o projetista tivesse estes padrões à sua disposição, de forma organizada, padronizada e integrada ao seu ambiente de desenvolvimento, e que, além disso, pudesse ser auxiliado na identificação de possíveis oportunidades para a sua aplicação.

2.4. Anti-Padrões

Embora os padrões disponíveis na literatura representem boas soluções para problemas recorrentes, KOENIG (1995) e COPLIEN (1997) ressaltam a importância de catalogar também o outro lado da moeda, o anti-padrão. Segundo KOENIG (1995), os anti-padrões descrevem soluções inadequadas para um problema que resultam em uma

situação ruim, ou então descrevem como sair de uma situação ruim e chegar a uma boa solução. COPLIEN (1997) ressalta que a presença de "bons" padrões em um sistema bem sucedido não é suficiente. É preciso mostrar que estes padrões estão ausentes de sistemas mal sucedidos, e que determinadas construções (anti-padrões) encontradas em sistemas mal sucedidos não estão presentes em sistemas bem sucedidos.

Nesta linha de detecção de construções problemáticas em projetos orientados a objetos, o estudo de anti-padrões aparece como uma recente atividade de pesquisa (BROWN et al., 1998). Um anti-padrão descreve uma solução para um problema recorrente que traz conseqüências negativas para o projeto. Um anti-padrão pode resultar da falta de conhecimento de uma solução melhor, ou ainda da aplicação de um padrão (teoricamente, uma boa solução) no contexto incorreto. Quando documentado de forma apropriada, um anti-padrão descreve o seu formato geral, as principais causas que levaram à sua aparição, os sintomas descrevendo como a sua presença pode ser reconhecida, as conseqüências que esta solução ruim pode gerar, e o que fazer para transformar esta solução em outra melhor.

Segundo BROWN et al. (1998), o conceito de anti-padrões é a primeira grande iniciativa de pesquisa em software orientado a objetos com foco em soluções negativas. Dada a freqüência de defeitos em sistemas e também de projetos fracassados, soluções problemáticas constituem um campo de estudo bastante rico. A pesquisa de anti-padrões tenta categorizar, nomear e descrever soluções ruins que ocorrem repetidamente na indústria, além de mostrar possíveis alternativas melhores que poderiam ter sido empregadas. O primeiro trabalho a formalizar alguns anti-padrões de projeto orientado a objetos é atribuído a (AKROYD, 1996).

De forma análoga aos padrões, os anti-padrões também são descritos de acordo com uma estrutura padronizada. Esta estrutura é um pouco diferente das propostas para os padrões, em função da própria natureza do anti-padrão. Ao invés de um problema e uma solução, um anti-padrão possui duas soluções: a primeira gera conseqüências negativas, enquanto que a segunda é uma migração ou reestruturação da primeira, visando eliminar, ou ao menos reduzir, os seus impactos negativos.

BROWN et al. (1998) propõem uma estrutura para a descrição de anti-padrões, composta pelas seguintes seções:

- **Nome:** de forma análoga aos padrões, a filosofia de dar nomes aos anti-padrões visa criar uma terminologia comum que facilite a comunicação entre os membros de uma equipe de desenvolvimento. Um anti-padrão também pode ser conhecido por outros nomes (sinônimos).

- **Forma Geral**: esta seção identifica as principais características do anti-padrão, podendo ter o apoio de diagramas. A nova solução resolve o problema descrito nesta seção.

- **Sintomas e Conseqüências:** lista de sintomas e conseqüências resultantes deste anti-padrão. Os sintomas fornecem indicações de como a presença do anti-padrão pode ser detectada. As conseqüências referem-se aos problemas que poderão resultar da aplicação da solução ruim.

- **Causas típicas**: identifica os principais motivos que levam ao aparecimento deste tipo de solução.

- **Exceções conhecidas**: uma determinada solução nem sempre é uma solução ruim, podendo ocorrer situações específicas onde esta solução possa ser aceita. Estas situações são descritas nesta seção.

- **Nova solução**: esta seção indica como a solução antiga deve ser reestruturada de forma a resolver as forças identificadas na seção "*Forma Geral*". A nova solução é apresentada na forma de passos.

- **Variações**: lista as possíveis variações deste anti-padrão. Se existirem soluções alternativas para este anti-padrão, elas são descritas nesta seção.

- **Soluções relacionadas**: se existirem outros anti-padrões relacionados ao que está sendo descrito, eles são listados nesta seção, explicitando-se os seus relacionamentos.

O número de Anti-Padrões catalogados ainda é pequeno, se comparado à quantidade de padrões disponíveis na literatura. KOENIG (1995) ressalta que é mais difícil aparecerem catálogos de anti-padrões do que de padrões. Isto porque, de uma maneira geral, as pessoas gostam de divulgar os seus sucessos e não os seus fracassos.

Em (BROWN et al., 1998), podemos encontrar uma relação de alguns anti-padrões, como por exemplo, o *"Blob"* (Figura 1).

Nome: *Blob*

Também conhecido por: Winnebago (AKROYD, 1996) ou God Class (RIEL, 1996)

Forma Geral: *Blob* é encontrado em projetos onde uma classe monopoliza o processamento e outras classes apenas encapsulam dados. Este anti-padrão é caracterizado por um diagrama de classes onde uma classe de controle complexa é circundada por várias classes simples. O maior problema desta solução é que a maior parte das responsabilidades do sistema estão alocadas em apenas uma classe.

Sintomas e Consequências:

- Uma classe com um grande número de atributos, operações, ou ambos. Uma classe com 60 ou mais atributos e operações, normalmente, indica a presença deste anti-padrão.
- Falta de coesão dos atributos e operações
- Uma classe de controle associada com objetos que apenas encapsulam dados.
- Esta solução compromete as vantagens de um projeto orientado a objetos, limitando a possibilidade de modificar o sistema sem afetar a funcionalidade de outros módulos.
- A classe *Blob*, normalmente, é muito complexa para ser reutilizada e testada. Pode ser ineficiente ou introduzir complexidade excessiva para a reutilização de subconjuntos de sua funcionalidade.

Causas Típicas:
- Projetistas não possuem conhecimento de heurísticas de projeto orientado a objetos, ou não possuem habilidade de abstração.
- Sistemas que evoluem de forma ad hoc, sem uma preocupação em definir, a priori, os componentes e suas interações.
- Tendência dos desenvolvedores em adicionar pequenos pedaços de funcionalidade em classes já existentes, ao invés de criar novas classes ou de revisar hierarquias de classes já existentes para uma melhor alocação de responsabilidades.

Exceções Conhecidas: Este anti-padrão é aceitável quando um sistema legado está sendo encapsulado.

Nova solução:
A nova solução consiste basicamente em reestruturar a antiga solução através da remoção do excesso de comportamento da classe centralizadora, distribuindo-o pelas classes mais simples com as quais ela se comunica ou criando novas abstrações. Esta remoção é feita seguindo os seguintes passos:
- Identificar ou categorizar conjuntos coesos de atributos e operações que representem contratos a partir da classe de controle que está centralizando dados e/ou comportamento.
- Alocar estes conjuntos em elementos adequados, criando novas abstrações, se necessário.
- Remover todas as associações indiretas ou redundantes.
- Migrar associações com classes derivadas para classes base comuns.

Variações:
RIEL (1996) identifica duas grandes formas em que o anti-padrão *Blob* se materializa, anti-padrão por ele denominado de "*classe Deus*": a forma comportamental onde um objeto centraliza o processo interagindo com classes simples que encapsulam apenas dados; e a forma de dados, onde um objeto encapsula dados compartilhados pela maioria dos outros objetos do sistema.

Figura 1: Anti-padrão *"Blob"*

Este anti-padrão corresponde a uma solução de projeto OO com forte degeneração para um estilo de projeto estruturado, onde as responsabilidades estão concentradas em um objeto, enquanto que a maioria dos outros objetos são utilizados como simples repositórios de dados, oferecendo apenas métodos de acesso aos seus atributos. Esta

solução compromete a facilidade de manutenção do sistema, devendo ser reestruturada de forma que as responsabilidades fiquem melhor distribuídas entre os objetos, isolando o efeito de possíveis modificações. A seguir, este anti-padrão é descrito de forma resumida, com o objetivo de exemplificar a descrição de um anti-padrão. Desta forma, os anti-padrões representam outro importante conhecimento de grande utilidade no sentido de auxiliar o projetista a tomar melhores decisões de projeto e evitar construções que possam trazer más conseqüências futuras. Da mesma forma que as heurísticas e padrões, seria altamente desejável que o projetista tivesse estes anti-padrões à sua disposição, de forma organizada, padronizada e integrada ao seu ambiente de desenvolvimento, e que, além disso, pudesse ser auxiliado na detecção de possíveis ocorrências destas construções em um projeto.

2.5. Relação entre Heurísticas, Padrões e Anti-Padrões

Enquanto heurísticas representam diretrizes genéricas para projetos orientados a objetos, um padrão é uma solução para um problema específico de projeto. As heurísticas normalmente motivam o aparecimento dos padrões (APPLETON, 1997). Podemos exemplificar esta relação através da heurística *"Minimize o acoplamento entre as classes"* (RIEL, 1996). Alguns padrões de projeto, como, por exemplo, o *"Facade"* (GAMMA et al., 1995), surgiram motivados por esta heurística. Este padrão fornece uma interface simples para um subsistema complexo, desacoplando-o de seus clientes, promovendo uma maior independência e portabilidade do subsistema. A Figura 2 ilustra a estrutura de um projeto antes da aplicação do padrão *"Facade"*, enquanto que a Figura 3 ilustra o resultado após a aplicação deste padrão.

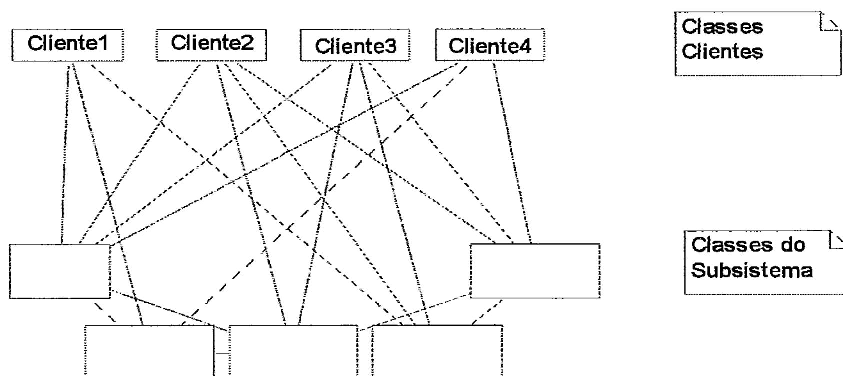


Figura 2: Estrutura de um projeto antes da aplicação do padrão *"Facade"*

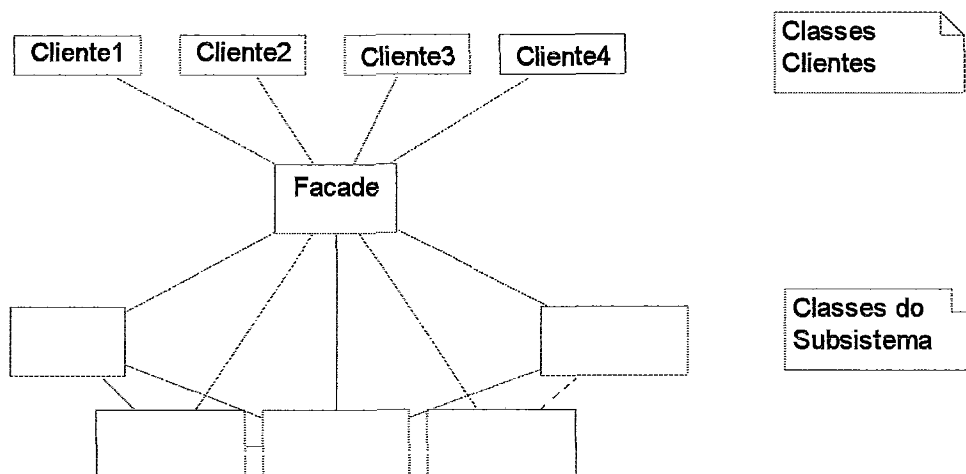


Figura 3: Estrutura de um projeto após a aplicação do padrão "*Facade*"

Um anti-padrão é visto como uma solução ruim de projeto que, normalmente, viola uma ou mais heurísticas de projeto. Em muitos casos, esta construção ruim pode ser reestruturada através da aplicação de um padrão de projeto. Uma vez que o número de anti-padrões disponíveis na literatura ainda é pequeno, uma possível alternativa para guiar o processo de busca de construções problemáticas em um software orientado a objetos seria utilizar os próprios catálogos de padrões. Estes catálogos, além de descreverem as boas soluções possíveis em um contexto, também discutem, informalmente, algumas soluções ruins que poderiam ter sido utilizadas. Estas soluções ruins podem ser, então, formalizadas e catalogadas de modo a compor uma base de anti-padrões de projeto orientado a objetos.

Como exemplo, podemos citar o padrão "*Singleton*" (GAMMA et al., 1995). Este padrão tem o objetivo de assegurar que uma classe tenha somente uma instância no programa, fornecendo um ponto de acesso global controlado à mesma. O desconhecimento deste padrão pode levar a soluções ruins como, por exemplo, a definição desta instância como uma variável global do sistema, ou como um atributo estático com visibilidade pública e com escopo de classe, ou seja, um atributo com um único valor compartilhado por todas as suas instâncias. Segundo GAMMA et al. (1995), em função de todos os clientes terem acesso direto a esta instância global, problemas de manutenção podem ocorrer, especialmente se o sistema for grande.

As heurísticas de projeto orientado a objetos podem ser outra fonte para a

detecção de problemas de projeto. Formas de violação conhecidas de uma determinada heurística podem ser interpretadas como anti-padrões. Conforme dito anteriormente, uma violação a uma heurística não corresponde necessariamente a um problema de projeto. Isto é perfeitamente compatível com a filosofia de anti-padrões, uma vez, que sua descrição permite identificar situações onde esta violação seria aceitável. Não se pode afirmar que a violação de uma heurística certamente corresponde a um problema de projeto, pois uma determinada construção pode ser decorrente de requisitos específicos de projeto como, por exemplo, eficiência, restrições de hardware/software, dentre outros.

Como exemplo, a definição de atributos de uma classe na sua área pública é um anti-padrão resultante da violação da heurística de que "*Todos os dados devem estar escondidos dentro de sua classe*" (RIEL, 1996). A violação desta heurística pode trazer conseqüências desastrosas em termos de manutenção do sistema, visto que fica muito difícil saber quais partes do sistema são dependentes destes dados. Uma vez que a representação da informação não está encapsulada, qualquer modificação pode gerar graves efeitos colaterais no restante do sistema.

Desta forma, podemos concluir que, apesar de aparecerem descritos separadamente na literatura, os conceitos de heurísticas, padrões e anti-padrões estão intimamente relacionados. Uma vez que estes conceitos são fundamentais para a construção de projetos orientados a objetos flexíveis e reutilizáveis, os projetistas muito se beneficiariam, caso estes conceitos estivessem disponíveis em uma base de conhecimento interligada e integrada ao seu ambiente de desenvolvimento.

2.6. Ferramentas de Suporte Existentes

As ferramentas CASE de apoio ao projeto orientado a objetos mais utilizadas hoje na indústria fornecem suporte à confecção de diversos diagramas (diagramas de classes, sequência, colaboração, transição de estados, componentes, processadores, por exemplo), assim como à geração de esqueletos de código a partir destes vários diagramas. Ferramentas como Rational Rose (RATIONAL, 1998), Paradigm Plus (PLATINUM, 1999), Select Enterprise (SELECT, 1999), entre outras, dão ênfase à

produção de diagramas corretos do ponto de vista das construções permitidas em cada diagrama. Entretanto, pouco ou nenhum suporte é dado à integração do conhecimento descrito nas seções anteriores, ou seja, heurísticas, padrões e anti-padrões, tão importante para a produção de bons projetos.

Com o objetivo de preencher esta lacuna, muitas pessoas estão trabalhando em métodos e ferramentas de suporte ao desenvolvimento de software incorporando este conhecimento. Algumas destas ferramentas são apresentadas nesta seção. Embora seja fato que a utilização efetiva deste conhecimento ainda dependa das habilidades intelectuais do desenvolvedor, algumas das experiências relatadas nesta seção demonstram a utilidade de tais ferramentas, não com o objetivo de automatizar totalmente a utilização de padrões e anti-padrões, mas sim como um valioso suporte para facilitar e promover a utilização destas habilidades intelectuais na sua plenitude.

As ferramentas encontradas na literatura podem ser classificadas em dois grandes grupos, conforme o seu foco principal de atuação:

- **Grupo 1:** apoio à engenharia reversa e reestruturação de aplicações, principalmente através do reconhecimento de padrões em sistemas, tendo como principal objetivo facilitar o entendimento e a manutenção de sistemas existentes e mal documentados.

- **Grupo 2:** apoio à utilização de padrões em novos sistemas, auxiliando na instanciação dos padrões e geração de código.

2.6.1. Apoio à Engenharia Reversa ou Reestruturação

Os principais objetivos das ferramentas deste grupo são:

- detectar construções típicas de projeto (padrões), principalmente a partir do código fonte de um sistema legado, visando facilitar o seu entendimento e manutenção.

- apoiar a reestruturação (*"refactoring"*) de aplicações através da substituição de determinados trechos do projeto por construções baseadas na aplicação de padrões de projeto.

Dentre os principais métodos e ferramentas disponíveis no *grupo 1*, podemos destacar as seguintes: *Pat* (KRÄMER e PRECHELT, 1996), *KT* (BROWN, 1997), *OMT Tool* (WINSEN, 1996), *DEPARS* (TE-WEI e MARTIN, 1997) e *Design Pattern Tool* (CINNÉIDE e NIXON, 1998).

2.6.1.1. Pat

Pat (KRÄMER e PRECHELT, 1996) é uma ferramenta que procura por instâncias de padrões de projeto em sistemas desenvolvidos em C++. O objetivo básico da ferramenta é facilitar o entendimento de sistemas grandes e/ou mal documentados através da detecção de padrões de projeto. A base fundamental para a busca automática efetuada pela ferramenta é representar em Prolog tanto os padrões como os elementos dos projetos onde estes padrões possivelmente foram instanciados, deixando para a máquina Prolog o trabalho de efetuar a busca propriamente dita. A representação em Prolog dos projetos existentes é realizada a partir da análise estrutural dos arquivos "header" dos programas em C++. Esta análise estrutural é feita pelo módulo de engenharia reversa da ferramenta CASE Paradigm Plus (PLATINUM, 1999).

O sistema limita-se a identificar cinco padrões estruturais de projeto orientado a objetos: "*Adapter*", "*Bridge*", "*Composite*", "*Decorator*" e "*Proxy*" (GAMMA et al., 1995). Esta detecção é baseada nas características estruturais de projeto obtidas pela ferramenta CASE, o que limita sobremaneira o processo, uma vez que ela não leva em conta nenhuma informação sobre as interações entre os objetos e a instanciação dos mesmos, fundamentais para a detecção de vários padrões, como, por exemplo, os de criação e os comportamentais (GAMMA et al., 1995). Além disto, as informações de projeto representadas pelos predicados propostos são muito reduzidas até mesmo para a detecção de padrões estruturais.

2.6.1.2. KT

KT é uma ferramenta desenvolvida por BROWN (1997) que realiza engenharia reversa de projetos OO a partir de código Smalltalk, tendo ainda a capacidade de detectar três padrões de projeto conforme descritos em (GAMMA et al., 1995): "*Composite*", "*Decorator*" e "*Template Method*".

A detecção das ocorrências de cada um destes três padrões é realizada por

algoritmos especificamente projetados para detectar estes três padrões em programas codificados em Smalltalk. Além de detectar estes três padrões, esta ferramenta gera, como resultado do processo de engenharia reversa, arquivos no formato da ferramenta Rational Rose (*"petal file"*), contendo os diagramas que representam a estrutura e as interações entre os objetos e classes no código Smalltalk analisado.

Em função de cada padrão ter o seu algoritmo de busca particular implementado diretamente em Smalltalk, a ferramenta limita sua atuação ao universo de programas Smalltalk, não oferecendo nenhum recurso para a detecção de padrões a partir de um modelo elaborado em uma ferramenta CASE de projeto.

2.6.1.3. OMT-Tool

A ferramenta *OMT-Tool* (WINSEN, 1996) foi criada com o propósito de apoiar a reestruturação de aplicações orientadas a objeto. O objetivo principal desta ferramenta é documentar ocorrências de padrões de projeto em sistemas existentes (programados em Smalltalk) e reorganizar estes sistemas através de operações de reestruturação disponíveis na ferramenta *"Smalltalk Refactoring Browser"*. As atividades envolvidas no processo de reestruturação de aplicações OO baseado nesta ferramenta são as seguintes:

- Importar o código Smalltalk, gerando um modelo correspondente à aplicação.
- Criar um novo projeto e definir todas as classes que fazem parte da aplicação como classes do projeto.
- Descobrir, através de análise efetuada por um especialista, os padrões que fazem parte da aplicação. A ferramenta não oferece suporte para a busca automática de padrões. Esta descoberta é apoiada por navegação nos diagramas de classe (estáticos) na notação OMT ou ainda no código fonte Smalltalk.
- Criar instâncias de padrões e ligá-las ao modelo gerado para a aplicação. À medida em que o usuário for descobrindo novos padrões, ele pode informar exatamente quais são os fragmentos da aplicação que fazem parte de uma nova instância do padrão, assim como o papel desempenhado por cada fragmento no padrão.
- Uma vez efetuada a ligação de uma instância de padrão a seus fragmentos, a

ferramenta pode verificar se, para uma determinada instância de padrão, todos os seus papéis e restrições foram atendidas. Além disso, a ferramenta oferece apoio na visualização das diversas instâncias de padrões que ocorrem na aplicação. Este apoio é baseado na utilização de cores e símbolos para representação visual dos diversos padrões.

- Reestruturar a aplicação, permitindo a mudança de nome das classes, remoção de classes, criação de subclasses, inclusão, exclusão e mudança de nome de operações, assim como diversas possibilidades de reestruturação dos atributos das classes. Esta atividade é realizada com o apoio de uma ferramenta de reestruturação desenvolvida pela Universidade de Illinois, baseada no trabalho de OPDYKE (1992) que foi incorporada ao ambiente.

2.6.1.4. DEPARS: Design Patterns Recognition System

O *DEPARS* (*Design Patterns Recognition System*) é um sistema cujo principal objetivo é reconhecer padrões de projeto em diagramas de classes elaborados utilizando-se uma ferramenta de modelagem orientada a objetos. Um protótipo do sistema (TE-WEI e MARTIN, 1997) foi desenvolvido em Smalltalk, com uma interface para a ferramenta de modelagem Argos (VERSANT, 1997)

O *DEPARS* utiliza uma estratégia baseada em conhecimento para reconhecer e detectar padrões em modelos de objetos. Todos os padrões são armazenados pelo sistema na forma de modelos. Cada modelo armazena o nome do padrão, informações estruturais, indicações de uso, impactos e referências para padrões similares ou alternativos. A cada classe ou relacionamento é atribuído um identificador. Dois modelos são conectados por uma ligação que representa suas diferenças estruturais. Estas diferenças são denominadas deltas. Um delta consiste de no máximo uma classe e exatamente uma relação.

TE WEI e MARTIN (1997) definem uma ontologia que caracteriza a estrutura de todos os modelos conhecidos e de todos os deltas entre os modelos. A ontologia começa de um modelo com apenas uma classe e expande-se à medida em que os modelos vão ficando mais complexos, servindo como conhecimento preliminar para o sistema, de modo a reduzir o espaço de busca a um determinado padrão.

Da mesma forma que as outras ferramentas de detecção de padrões citadas anteriormente, o *DEPARS* também reduz o seu escopo de busca às informações presentes no diagrama de classes, não levando em consideração informações sobre a dinâmica de colaboração entre os objetos, o que, conforme dito anteriormente, limita o processo de detecção aos padrões estruturais.

2.6.1.5. Design Pattern Tool (DPT)

Design Pattern Tool (CINNÉIDE e NIXON, 1998) é uma ferramenta que realiza transformações em programas Java com o intuito de introduzir padrões de projeto do tipo *"creational"* (GAMMA et al., 1995). Para cada um dos cinco padrões (*"Abstract Factory", "Factory Method", "Builder", "Singleton" e "Prototype"*) foi definido um algoritmo de transformação com o objetivo de reestruturar uma determinada construção do programa, onde o padrão poderia ter sido aplicado, em outra, resultante da aplicação do padrão. Esta transformação é realizada diretamente nas árvores resultantes da análise do programa que esteja sofrendo a modificação.

Como exemplo destas transformações, suponha a instanciação de um objeto de uma classe *"Product"* dentro de um método de uma classe *"Creator"*. Este fragmento de código pode ser transformado através da aplicação do padrão *"Factory Method"*. A Figura 4 ilustra um exemplo de especificação desta transformação.

```
ApplyFactoryMethod(Class creator, Class product) {
        addInterface(product, "abs"+product.name());
        creator.renameType(product.name(), "abs"+product.name());
        for all Constructor c in Class product do {
          Method newMethod("create" + product.name());
          newMethod.returnType = "abs" + product.name();
          newMethod.paramList = c.paramList;
          newMethod.body = ("return new P(" + c.paramList + ");");
          creator.addMethod(newMethod);
        }
        creator.replaceObjectCreations(product, "create" + product.name());
}
```

Figura 4: Exemplo de transformação do DPT

Para aplicar esta transformação, o programador deve especificar a classe que cria o produto (*"Creator"*) e a classe do produto (*"Product"*). O nome do novo *"Factory Method"* é gerado automaticamente, colocando-se o sufixo *"create"* ao nome da classe *"Product"*. Os métodos públicos da classe *"Product"* são abstraídos em uma interface que, através de uma relação de realização (*AddInterface*), é adicionada a esta classe. Todas as referências para a classe *"Product"* na classe *"Creator"* são, então, atualizadas para esta nova interface. Para cada construtor da classe *"Product"*, um método similar é adicionado à classe *"Creator"*, retornando uma instância da classe *"Product"*. Finalmente, todas as instanciações de objetos da classe *"Product"* na classe *"Creator"* são substituídas por estes novos métodos.

O foco desta ferramenta não está na detecção das construções onde os padrões poderiam ter sido aplicados, e sim na substituição das mesmas, após terem sido manualmente descobertas, pelos respectivos padrões. Esta substituição é feita de forma a manter o comportamento original do programa.

2.6.2. Apoio ao Desenvolvimento com Padrões

Dentre as principais ferramentas disponíveis no *grupo 2,* isto é, ferramentas que fornecem suporte à utilização de padrões na construção de novos sistemas, podemos destacar as seguintes: um gerador automático de código a partir de padrões de projeto – *IBM* (BUDINSKY et al., 1996), a proposta oriunda de um grupo da Universidade de Waterloo de uma ferramenta para definição e aplicação formal de padrões de projeto (ALENCAR et al., 1995), um ambiente de suporte ao desenvolvimento de sistemas baseado em padrões (FLORIJN et al., 1997), e a ferramenta *Framework Studio* (BLUEPRINT, 1999) que compreende um catálogo com diversos padrões disponíveis na literatura, fornecendo suporte à sua instanciação em tempo de projeto.

2.6.2.1. Gerador automático de código a partir de padrões de projeto - IBM

Conforme descrito na seção 2.3, padrões de projeto são abstrações que descrevem a solução para um problema específico de projeto devendo, portanto, ser implementados a cada vez em que forem aplicados. Os projetistas devem fornecer nomes (específicos

do domínio da aplicação) aos elementos participantes de um padrão (classes, interfaces, atributos, operações) e implementar as declarações e definições destes elementos de acordo com o prescrito no padrão. Além disso, podem existir várias considerações a respeito das conseqüências da aplicação de um padrão, resultando em possíveis variantes de implementação, o que pode fazer com que os desenvolvedores dupliquem esforços a cada nova aplicação de um padrão.

Em face deste problema, BUDINSKY et al. (1996) apresentam uma ferramenta para geração automática de código baseada em padrões de projeto a partir de informações fornecidas pelo usuário. A ferramenta incorpora também uma versão hipertexto do livro "*Design Patterns – Elements of Reusable Object Oriented Software*" (GAMMA et al., 1995) fornecendo, de forma integrada, uma referência on-line aos padrões e uma ferramenta de apoio à geração de código. Os objetivos fundamentais da ferramenta são:

- facilitar a transição da descrição de um padrão para uma implementação particular.
- facilitar repetidas aplicações de um padrão

Através de um navegador Web, o usuário tem acesso aos padrões disponíveis. Os padrões são descritos segundo o formato *GoF* (GAMMA et al., 1995). Cada seção do padrão corresponde a uma página de exibição. A ferramenta disponibiliza ainda uma seção adicional para cada padrão denominada "*Geração de Código*". Através desta página, a ferramenta permite que o usuário gere o código correspondente à aplicação do padrão em questão.

A geração automática de código a partir de padrões de projeto permite que os usuários possam ver de que maneira determinados conceitos de projeto são mapeados em código que implementa um padrão, podendo, ainda, visualizar como a escolha de diferentes implementações afeta o código produzido. Uma vez gerado, o usuário pode colocar o código para funcionar imediatamente, amenizando o trabalho repetitivo de aplicar diversas vezes um determinado padrão.

A ferramenta fornece suporte apenas para os padrões de projeto descritos em (GAMMA et al., 1995), não permitindo que o usuário incorpore novos padrões ao ambiente. Outra deficiência é o fato da ferramenta funcionar de forma isolada, ou seja,

ela apenas gera o código especificado pelo usuário através das suas páginas de geração de código, não se integrando a uma ferramenta CASE de projeto. Desta forma, a instanciação dos padrões, ao invés de ocorrer em um nível de abstração de projeto, isto é, gerando elementos no ambiente de projeto, limita-se a gerar trechos de código do projeto, ficando a cargo do usuário a tarefa de integração destes elementos no sistema como um todo.

2.6.2.2. Universidade de Waterloo – Ferramenta para definição e aplicação formal de padrões de projeto

Esta proposta (ALENCAR et al., 1995) consiste em uma ferramenta para a definição e aplicação formal de padrões de projeto. Padrões de projeto têm sido utilizados a partir de descrições informais baseadas fortemente em texto o que, segundo os autores, dificulta o suporte de ferramentas para uma utilização mais efetiva de padrões. Um modelo formal para os padrões de projeto permite uma definição precisa dos passos que devem ser seguidos na instanciação de um padrão, servindo como base para ferramentas de suporte e geração de código. Este modelo é baseado em dois conceitos: *ADV* (Abstract Data View) e *ADO* (Abstract Data Object).

Um *ADO* é um objeto que não possui nenhum contato direto com o mundo exterior. Como todo objeto, um *ADO* possui estado e uma interface pública que pode ser utilizada para consultar ou modificar o seu estado. Um *ADV* é um *ADO* com suporte para o desenvolvimento de visões de *ADO*s, onde uma visão poderia incluir uma interface com usuário ou uma adaptação da interface pública de um *ADO*.

A separação entre visões e objetos permite utilizar vários *ADV*s para criar diferentes visões de uma coleção única de *ADO*s. Neste caso, tanto os *ADO*s como suas visões associadas devem ser consistentes. Por exemplo, um *ADO* relógio pode estar associado a um *ADV* digital e a um *ADV* analógico. A estrutura de todo *ADV* ou *ADO* é subdividida em três seções:

- Declarações: descrição de todos os elementos que compõem o objeto incluindo, funções, atributos, ações.
- Propriedades Estáticas: define todas as propriedades que não afetam o estado do objeto.

- Propriedades Dinâmicas: definem como os valores de estado e atributos de um objeto são modificados ao longo do seu ciclo de vida.

A definição formal de um padrão de projeto é formada por duas partes:

- *Parte independente da linguagem*: define claramente as características do padrão, devendo conter quatro elementos essenciais: um nome, o contexto onde o padrão é aplicável, um detalhamento da solução proposta e as conseqüências da sua aplicação.

- *Parte dependente da linguagem:* descreve o resultado da aplicação do padrão como uma representação formal baseada nas estruturas *ADV* e *ADO* descritas anteriormente.

Baseado nestas duas partes, todo padrão é descrito segundo o esquema ilustrado na Figura 5. Um protótipo de uma ferramenta de suporte à aplicação de padrões foi desenvolvido, permitindo a instanciação de padrões através de uma interface Web. O usuário seleciona o padrão que deseja instanciar e preenche informações em uma série de janelas, onde cada janela corresponde a um passo da seqüência que deve ser seguida para instanciar o padrão selecionado. Após preencher todas as informações, o usuário recebe como resposta a seção *"Product text"* do padrão, podendo gerar código C++ a partir deste esquema gerado. Esta ferramenta, assim como a descrita na seção anterior, também não é integrada a uma ferramenta CASE de projeto.

Operator <Nome do Padrão>	
Objective:	descrição do objetivo do padrão.
Parameters:	elementos externos utilizados na definição do padrão.
Subtasks:	descrição do padrão em construtores primitivos.
Consequences:	como o padrão suporta o seu objetivo.
Product Text:	especificação do padrão dependente de linguagem.
End Operator	

Figura 5: Descrição formal de um padrão

2.6.2.3. Universidade de Utrecht – Ambiente de suporte ao desenvolvimento de sistemas baseado em padrões

FLORIJN (1997) propõe um ambiente de suporte ao desenvolvimento de sistemas baseado em padrões. Este ambiente deve fornecer três visões mutualmente consistentes:

- *Nível padrão*: onde seria possível colocar padrões nos sistemas, conectá-los uns aos outros, modificar uma ocorrência de um padrão por outro, etc.
- *Nível projeto*: permitir a visualização da estrutura de projeto em termos de classes, métodos, associações, heranças, etc.
- *Nível código*: edição de código em alguma linguagem de programação.

Um protótipo deste ambiente foi criado com o objetivo de permitir o desenvolvimento de software orientado a objetos com padrões. O ambiente fornece suporte ao uso de padrões tanto no desenvolvimento de novas aplicações, como na reestruturação de aplicações orientadas a objetos já existentes (através da ferramenta *OMT-Tool*, descrita na seção 2.6.1.3.). Uma das idéias fundamentais deste ambiente é um metamodelo para representação dos padrões. Este modelo denominado "*Modelo de Fragmentos*" consiste em representar cada elemento relevante de um padrão por um fragmento. Os fragmentos pertencentes ao mesmo padrão de projeto são relacionados entre si por "*papéis*". Por exemplo, os fragmentos correspondentes aos métodos de uma classe estão a ela ligados por um papel "*método*". Cada fragmento possui um número arbitrário de papéis, que são preenchidos por outros fragmentos.

A ferramenta permite a instanciação de um padrão de projeto dentre os vários armazenados em um repositório de padrões. Este processo envolve a produção de um modelo correspondente aos fragmentos e papéis desempenhados por cada fragmento, conforme o modelo de fragmentos correspondente ao padrão desejado. A partir daí, o usuário pode modificar o modelo de acordo com suas necessidades específicas, sendo que a ferramenta dispõe ainda de um mecanismo para verificar se estas modificações continuam respeitando possíveis restrições de um padrão em particular.

Ao contrário de outras ferramentas, a inclusão de novos padrões no sistema é bem simples, bastando descrever o modelo de fragmentos correspondente a cada novo

padrão a ser incluído. Entretanto, a definição dos padrões é baseada apenas nas características estruturais, ou seja, nas classes, atributos, métodos e relacionamentos estruturais entre as classes, deixando de fora informações correspondentes à dinâmica de colaboração entre os objetos que compõem o padrão.

2.6.2.4. Framework Studio

Framework Studio (BLUEPRINT, 1999) é um produto que pode ser acoplado à ferramenta CASE Rational Rose (RATIONAL, 1998), e tem como objetivo fornecer ao projetista um catálogo de diversos padrões coletados da literatura tais como (GAMMA et al., 1995), (BUSCHMANN et al., 1996), (MOWBRAY e MALVEAU, 1997). A partir deste catálogo, o projetista pode escolher o padrão mais apropriado a um determinado problema, e instanciá-lo na ferramenta CASE.

O processo de instanciação é feito através da definição de cada participante do padrão, onde o projetista indica como as classes, atributos e métodos do modelo devem ser mapeados em cada elemento do padrão. Este processo de instanciação é limitado ao mapeamento da parte estrutural dos participantes do padrão. A parte dinâmica deve ser inserida manualmente pelo projetista.

2.7. Conclusões

Os métodos e ferramentas de apoio ao desenvolvimento orientado a objetos comercialmente disponíveis oferecem pouco ou nenhum suporte para a produção de software flexível e reutilizável. Como consequência, vários sistemas desenvolvidos com tecnologia orientada a objetos já sofrem com problemas de manutenção. Os projetistas, principalmente os iniciantes, carecem de uma base de conhecimento integrada às suas ferramentas de desenvolvimento que disponibilize informações, como heurísticas e soluções que sabidamente possam trazer conseqüências boas (padrões) ou ruins (anti-padrões), e permita utilizá-las de forma inteligente no projeto.

Avaliando as ferramentas que tentam incorporar suporte automatizado à utilização deste conhecimento, é possível verificar que nenhuma dá um tratamento integrado à esta questão. A ênfase de pesquisa concentra-se nos padrões de projeto (soluções boas)

visando principalmente a compreensão em um nível maior de abstração (padrões) de um projeto orientado a objetos, ou à instanciação dos mesmos em um novo projeto.

O conceito de administração de um repositório aberto e extensível de padrões está presente apenas nas propostas de FLORIJN (1997) e da BLUEPRINT (1999). Na parte de reconhecimento de padrões, a grande maioria das propostas limitou-se ao suporte a padrões de projeto (GAMMA et al., 1995), em especial aos padrões estruturais. As ferramentas fornecem suporte fraco ou inexistente ao reconhecimento de padrões comportamentais, isto em função de trabalharem apenas com a definição estática das classes, isto é, com informações disponíveis a partir de diagramas de classes. Algumas ferramentas fornecem suporte à geração de código a partir dos padrões, como as propostas de BUDISNKY et al. (1996) e ALENCAR et al. (1995). O suporte a um catálogo organizado de heurísticas de projeto orientado a objetos também não foi encontrado nas ferramentas analisadas.

Em função da área de anti-padrões ser, relativamente, mais recente e também do fato de que os anti-padrões ainda não despertam a mesma atenção hoje dispensada aos padrões, talvez por causa dos aspectos psicológicos associados descritos na seção 2.4, o suporte de ferramentas ao conceito de anti-padrões hoje disponível é praticamente nulo. Entretanto, sabemos que muito podemos aprender com os erros cometidos por nós ou por outros. O projetista poderia ser muito beneficiado com o suporte de um catálogo com estes erros comuns materializados sob a forma de construções ruins, e de recursos de detecção destas construções problemáticas em um projeto.

A Tabela 1 mostra um resumo comparativo das características de suporte a heurísticas, padrões e anti-padrões fornecidas por estas propostas.

	PAT	*KT*	*OMT Tool*	*Depars*	*DPT*	*IBM*	*Waterloo*	*Florijn*	*Framework Studio*
Administração de um repositório de heurísticas									
Administração de um repositório de padrões						somente consulta		✓	✓
Administração de um repositório de anti-padrões									
Integração dos conceitos de heurísticas, padrões e anti-padrões									
Detecção de padrões estruturais	✓	✓		✓					
Detecção de padrões comportamentais e de instanciação									
Detecção de anti-padrões									
Instanciação de padrões em projeto							✓	✓	✓
Instanciação de padrões em código						✓	✓		
Suporte à reestruturação de projeto baseada em padrões			✓		✓				

Tabela 1: Quadro resumo do suporte a padrões, anti-padrões e heurísticas

Capítulo 3

Uma Arquitetura de Apoio para a Análise de Modelos Orientados a Objetos

3.1. Introdução

O capítulo anterior descreveu a importância dos conceitos de heurísticas, padrões e anti-padrões dentro do contexto de desenvolvimento de sistemas orientado a objetos. Este capítulo apresenta uma proposta que visa disponibilizar estas informações para o projetista de forma organizada, padronizada e integrada com uma ferramenta de projeto, permitindo a detecção de construções típicas, tanto as boas (padrões) como as ruins (anti-padrões). O principal objetivo desta detecção é fornecer suporte tanto para a reestruturação de sistemas OO legados, como também para a avaliação de sistemas OO ainda em desenvolvimento. A detecção de construções problemáticas (anti-padrões) permite que determinadas construções que comprometam a futura expansão ou modificação dos mesmos possam ser detectadas e substituídas por outras mais adequadas, ou seja, podemos detectar pontos do sistema que devam ser modificados, de modo a torná-lo mais flexível e reutilizável. A detecção de construções típicas (padrões) permite que o projeto do sistema seja entendido em um nível maior de abstração, além de permitir uma avaliação sobre a utilização destes padrões no projeto.

A tarefa de identificar padrões e anti-padrões em um projeto orientado a objetos é muito difícil de ser realizada manualmente em função de vários motivos, dentre os quais podemos destacar:

- os sistemas legados que despertam interesse e necessidade em sua reestruturação, normalmente, correspondem a sistemas de médio/grande porte, onde a busca manual por problemas se torna impraticável.
- na maioria das vezes, a única fonte segura de informação sobre o projeto está no código fonte, ou seja, os modelos, quando existem, estão desatualizados ou são superficiais demais para a detecção da maior parte dos problemas. A análise manual

do código fonte limita sobremaneira o escopo dos problemas que podem ser detectados de forma economicamente viável.

- os desenvolvedores muitas vezes não sabem que tipo de construção procurar. A existência de uma base de construções de projeto, englobando tanto padrões como anti-padrões, pode fornecer um suporte valioso nesta busca.

Este capítulo apresenta uma arquitetura destinada a apoiar a detecção de padrões e anti-padrões em projetos orientados a objetos. Esta arquitetura é composta pelos seguintes componentes (Figura 6):

- **Extrator de projeto**: este componente tem como objetivo extrair informações de projeto a partir de um código fonte escrito em uma linguagem orientada a objetos como, por exemplo, C++, Java, Smalltalk, armazenando-as automaticamente em uma ferramenta CASE orientada a objetos. As informações capturadas por este componente seguem um metamodelo para representação de projetos orientados a objetos, descrito na seção 3.2.

- **Gerador de fatos:** este componente é responsável pela geração de uma base de fatos correspondente às construções existentes no projeto, de acordo com o metamodelo para representação de projetos orientados a objetos. O resultado é uma base que deve representar, de modo independente da ferramenta CASE utilizada para modelar o projeto, todas as construções de projeto encontradas, como, por exemplo, as classes, seus atributos, operações, relacionamentos, interações entre objetos, enfim, todos os fatos representativos de um projeto orientado a objetos que servirão de fonte para a detecção de padrões e anti-padrões.

- **Captura de Conhecimento:** este componente é responsável pela alimentação da base de conhecimento sobre heurísticas, padrões e anti-padrões, que servirá como fonte de consulta para os projetistas. Este componente deve permitir a catalogação de informações detalhadas sobre heurísticas, padrões e anti-padrões, bem como os relacionamentos existentes entre eles. Além disso, esta base deve definir todas as regras que caracterizam um padrão ou anti-padrão, com o objetivo de permitir a sua detecção em um projeto orientado a objetos.

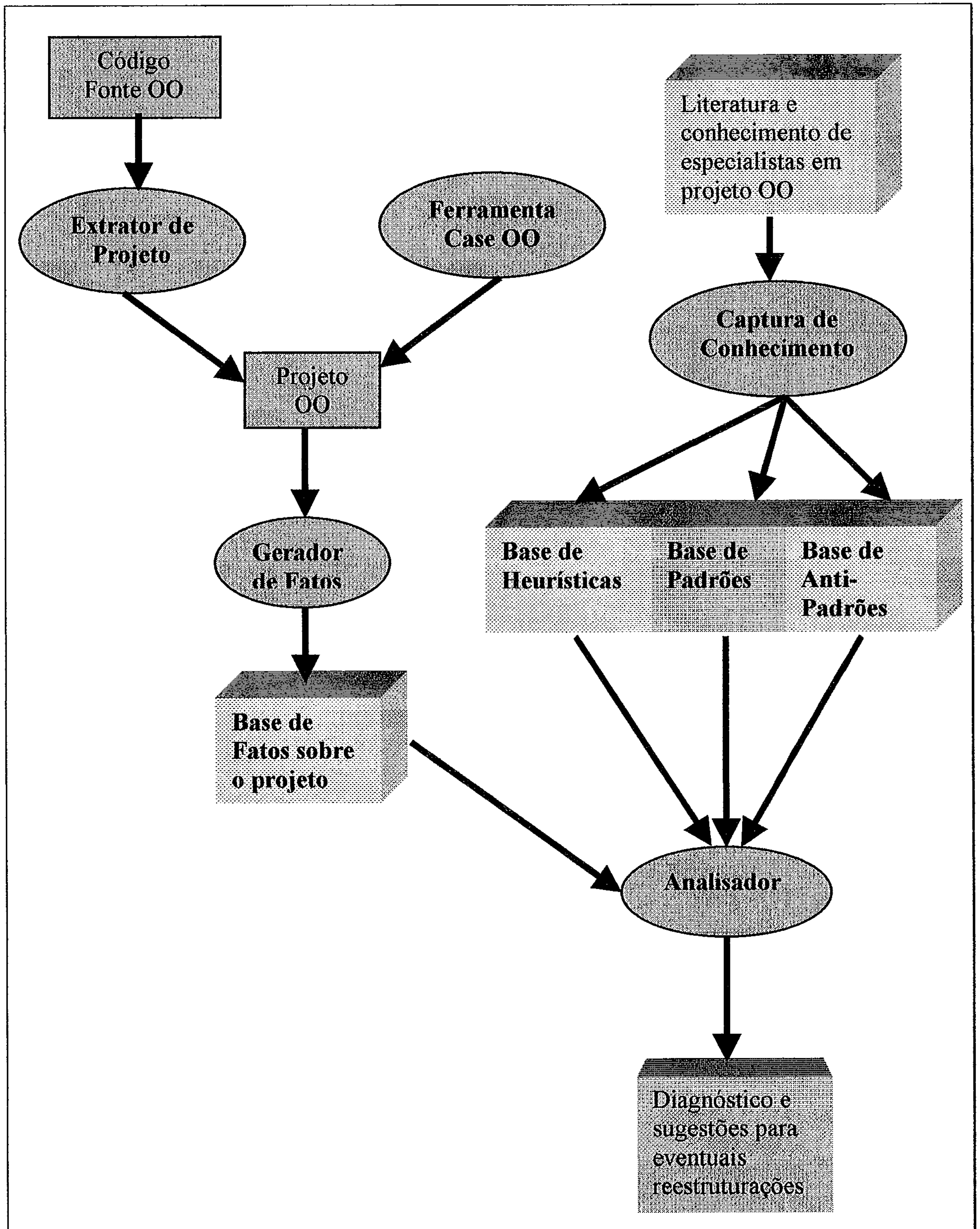


Figura 6: Componentes da abordagem

- **Analisador**: este componente tem como objetivo analisar a base de fatos de um projeto a partir do conhecimento armazenado na base de conhecimento sobre heurísticas, padrões e anti-padrões. Esta análise tenta detectar os fragmentos de projeto que potencialmente poderão gerar problemas de manutenção e reutilização futura, indicando caminhos para a transformação destas construções em outras mais flexíveis. Além da detecção de anti-padrões, este componente deve possibilitar também a detecção de padrões utilizados no projeto.

O capítulo está organizado da seguinte forma: a seção 3.2 descreve o metamodelo utilizado para representar as construções de um projeto orientado a objetos. A seção 3.3 descreve o componente *Extrator de Projeto*. Na seção 3.4, o componente *Gerador de Fatos* é descrito. A seção 3.5 detalha as características do componente *Captura de Conhecimento*. A seção 3.6 apresenta o componente *Analisador*. Finalizando o capítulo, a seção 3.7 apresenta um resumo e conclusões sobre a arquitetura proposta.

3.2. Metamodelo

A captura de informações a partir do código fonte escrito em uma linguagem orientada a objetos e a geração da base de fatos descrevendo o projeto devem ser efetuadas levando-se em consideração as informações relevantes para a detecção dos padrões e anti-padrões. Estas informações compõem um metamodelo que descreve os elementos básicos a partir dos quais um projeto orientado a objetos é construído.

O metamodelo descrito neste trabalho é resultado de uma compilação do metamodelo semântico definido pela linguagem de modelagem unificada – UML (OMG, 1998) e de outros trabalhos na área de metamodelos para a modelagem orientada a objetos, tais como os descritos em (DEMEYER et al., 1998) e (MAUGHAN e AVOTINS, 1998). O metamodelo define as principais entidades de um projeto OO (pacote, classificador, atributo, operação, parâmetro), relacionamentos entre elas (dependência, realização, herança), assim como elementos representando informações correspondentes à implementação dos métodos como, por exemplo, a instanciação de um objeto, a destruição de um objeto, a invocação de uma operação e o acesso a um determinado atributo. Os elementos que descrevem a estrutura estática dos elementos de um projeto OO foram definidos a partir do metamodelo da UML (OMG, 1998), enquanto que os elementos correspondentes à parte dinâmica foram obtidos a partir das

definições dos outros trabalhos citados, uma vez que o metamodelo da UML não contempla informações sobre a implementação de métodos, como a invocação de operações, por exemplo.

A Figura 7 ilustra os elementos estruturais do metamodelo na notação UML (OMG, 1998), com as construções fundamentais da visão estática de um projeto orientado a objetos.

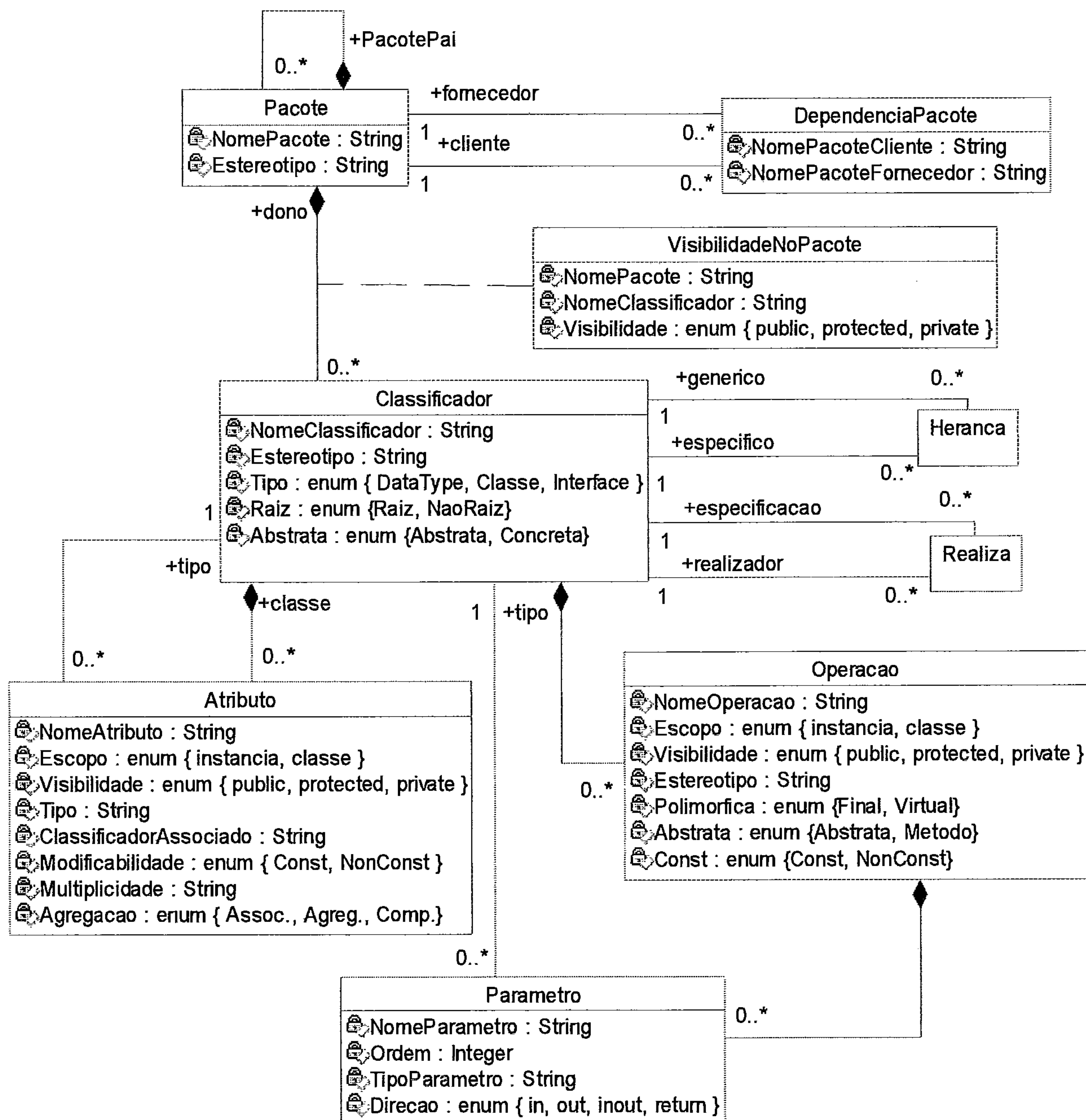


Figura 7: Elementos básicos do metamodelo

3.2.1. Pacotes

Um *pacote* é um mecanismo geral para organizar os elementos de um modelo em grupos. *Pacotes* bem projetados resultam do agrupamento de elementos que estão semanticamente próximos e que apresentem uma tendência de mudarem juntos. Desta forma, *pacotes* tendem a apresentar um fraco acoplamento entre si, uma forte coesão e um acesso bastante controlado aos elementos que o compõem (BOOCH et al., 1999). A análise das interdependências entre os *pacotes* pode trazer à tona várias construções problemáticas.

Dentre estas construções problemáticas, podemos citar uma descrita em (MARTIN, 1995) e ilustrada na Figura 8. Esta construção corresponde a uma dependência circular entre dois pacotes. Esta dependência circular pode ser direta (A depende de B e vice-versa) ou indireta (A depende de B; B depende de C; e C depende de A). Dizemos que um pacote A é dependente de um pacote B quando pelo menos um elemento de A é dependente de um elemento de B (RUMBAUGH et al., 1999). Segundo (MARTIN, 1995), é essencial manter uma estrutura de projeto onde não existam ciclos entre pacotes. Dependências cíclicas entre pacotes devem ser eliminadas, pois impedem que os mesmos possam ser desenvolvidos e modificados de forma independente.

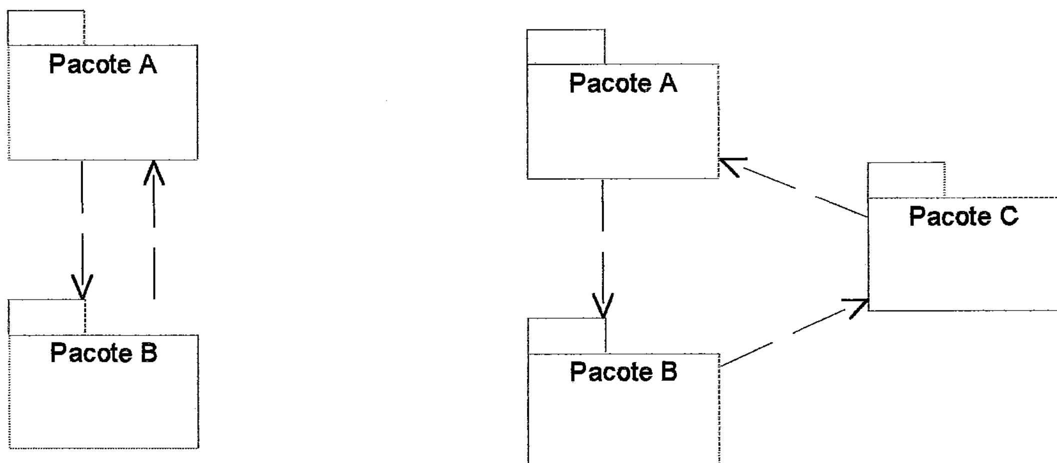


Figura 8: Dependência circular entre pacotes.

3.2.2. Classificadores

Um pacote pode conter outros pacotes e classificadores. Um *classificador* corresponde à definição de uma *classe, interface* ou *tipo básico de dados* (OMG, 1998).

Uma *classe* é a descrição de um conjunto de objetos que compartilham os mesmos *atributos, operações, métodos, relacionamentos e semântica*. Uma *classe* pode representar um conceito do domínio do problema que está sendo modelado, como também pode corresponder a elementos específicos do mundo computacional (universo de solução). Uma *interface* é a declaração de uma coleção de operações que, juntas, definem um serviço oferecido por um *classificador* que realize esta *interface*. *Interfaces* não podem possuir atributos, associações ou implementações de operações. Uma *classe* pode realizar uma ou mais *interfaces*, devendo fornecer uma implementação para todas as *operações* definidas em cada uma das interfaces realizadas. Um *tipo básico de dados* descreve um conjunto de valores sem identidade. *Tipos básicos de dados* correspondem normalmente a números, cadeias de caracteres, data e hora. Tipos básicos definidos pelo usuário são definidos como enumerações (RUMBAUGH et al., 1999).

Um *classificador* pode ser *raiz* de uma hierarquia de classes, o que faz com que ele não possa ser derivado de nenhum outro *classificador*. Um *classificador concreto* é aquele que pode ser instanciado, enquanto que um *abstrato* não permite a geração de instâncias. Uma *interface* é, por definição (RUMBAUGH et al., 1999), um *classificador abstrato*, enquanto que uma *classe* pode ser *concreta, abstrata* (se possuir ao menos uma operação abstrata, ou se não disponibilizar um método construtor na sua área pública), ou *totalmente abstrata* (todas as operações declaradas na classe são abstratas).

Um *classificador* pode ainda ter um *estereótipo*. Um *estereótipo* serve para adicionar semântica a elementos que tenham a mesma forma, mas possuam objetivos diferentes. Algumas construções problemáticas podem ser capturadas com base em informações fornecidas pelos *estereótipos*. Por exemplo, JACOBSON et al. (1999) definem *estereótipos* básicos para as classes de um sistema, dentre os quais podemos citar *"Limítrofe", "Controle", "Entidade"*. Além de definir estes *estereótipos*, JACOBSON et al. (1999) definem regras de dependência entre estes elementos que devem ser respeitadas para que se obtenha um projeto flexível e reutilizável. Um elemento com *estereótipo "Entidade"*, por exemplo, não deve ser dependente de um elemento com *estereótipo "Limítrofe"*, sendo desejável a existência de dependências apenas no sentido inverso, ou seja, elementos "Limítrofes" dependentes de elementos "Entidade".

3.2.3. Visibilidade de elementos dos pacotes

Um pacote define a visibilidade dos seus elementos para outros pacotes. Elementos com visibilidade *"private"* são visíveis apenas dentro do pacote, enquanto que elementos com visibilidade *"public"* são visíveis para outros pacotes. Ao organizar os elementos de um projeto em pacotes, controlando o acesso a estes elementos através de relações de dependência entre os pacotes, o projetista pode gerenciar a complexidade de um projeto com um número grande de abstrações. O conceito de visibilidade dentro de um pacote cria um outro nível de encapsulamento especialmente importante no contexto de sistemas grandes. Alguns padrões de projeto, como o *"Facade"* (GAMMA et al., 1995), por exemplo, podem ser utilizados para minimizar a exposição de classes na interface pública do pacote, uma vez que, idealmente, as dependências entre pacotes devem ocorrer apenas através de interfaces, e não de implementações (D'SOUZA e WILLS, 1999).

3.2.4. Atributos

Classificadores podem definir *atributos* e *operações*. O elemento *Atributo* do metamodelo representa um atributo simples ou uma associação com um *classificador* (pseudo-atributo). Um atributo simples é um elemento de uma classe que descreve um conjunto de valores que instâncias desta classe devem manter. Normalmente, um atributo simples é definido para valores sem identidade (tipos básicos de dados), sendo semanticamente equivalente a uma associação de composição (RUMBAUGH et al., 1999). Neste metamodelo, o elemento *Atributo* é utilizado tanto para capturar a existência destes atributos simples, como dos pseudo-atributos. Pela definição da UML (OMG, 1998), um pseudo-atributo representa o papel desempenhado por um classificador em um relacionamento. Para exemplificar esta diferença, suponha o fragmento de modelo descrito na Figura 9. Neste modelo, Fita é um classificador com dois atributos associados: *número* (correspondente ao atributo simples definido diretamente em Fita) e *oFilme* (pseudo-atributo resultante da associação com o classificador Filme).

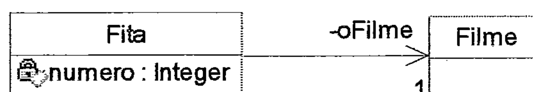


Figura 9: Exemplo de atributo e pseudo-atributo

A UML (OMG, 1998) define três tipos básicos de relacionamentos: associação, composição e agregação. Agregação define um relacionamento todo-parte entre um agregado e suas partes constituintes. Composição é uma forma mais forte de agregação, onde um elemento pode ser parte de apenas um todo e além disso, o todo tem responsabilidade de criação e destruição das suas partes. Associação é um relacionamento que envolve a conexão entre instâncias das classes associadas sem a existência de uma semântica do tipo todo-partes. Quando o relacionamento entre dois classificadores A e B for navegável de A para B, isto implica na existência de um pseudo-atributo em A com o nome do papel exercido por B no relacionamento. O mesmo ocorre, de forma análoga, quando o relacionamento for navegável de B para A. No exemplo da Figura 9, o relacionamento é navegável apenas de Fita para Filme.

Um atributo pode ter visibilidade *"private"*, *"protected"* ou *"public"* na definição do classificador. O escopo do atributo pode ser de *"instância"* ou de *"classe"*. O escopo de instância significa que cada instância tem um espaço individual para o valor deste atributo, enquanto que o escopo de classe faz com que todas as instâncias compartilhem o mesmo espaço para um determinado atributo. O escopo do atributo pode ser importante na detecção de algumas construções problemáticas. Segundo (RUMBAUGH et al. 1999), atributos com escopo de classe fornecem valores globais para toda a classe e devem ser utilizados com muito cuidado ou evitados totalmente, uma vez que eles violam o espírito de orientação a objetos. A combinação de um atributo com escopo de classe com visibilidade pública é um exemplo de uma construção ruim de projeto que poderia ser substituída pela aplicação de um padrão de projeto conhecido como *"Singleton"* (GAMMA et al., 1995).

O valor de um atributo pode ser fixo (*"Const"*) ou variável (*"NonConst"*). A multiplicidade de um atributo indica se o atributo possui um ou múltiplos valores. Todo atributo está associado a um classificador que define o seu tipo. Para o caso onde o atributo seja multivalorado, normalmente eles são armazenados em classes coleção (BOOCH et al., 1999), como "vector", "list", "set", dentre outras. O atributo tipo da entidade "Atributo" do metamodelo indica a classe coleção utilizada para armazenar atributos multivalorados. Para atributos com um único valor, os elementos Tipo e ClassificadorAssociado têm o mesmo significado.

3.2.5. Operações

Uma operação é a especificação de um serviço que pode ser requisitado a um objeto. Uma operação possui uma assinatura que descreve os parâmetros possíveis na sua invocação, incluindo valores de retorno. Um método é a implementação de uma operação, especificando o algoritmo ou procedimento que produz os resultados da operação. No metamodelo, os conceitos de operação e método são definidos pela mesma entidade "Operação". A diferenciação entre os dois conceitos é feita por uma propriedade (*"Abstrata"*) que indica se o elemento corresponde a uma definição abstrata (operação) ou a uma implementação (método).

De forma análoga aos atributos, as operações tem uma visibilidade dentro do classificador e também um escopo (instância ou classe). Uma operação pode ter um método que a implemente em uma classe, de tal forma que não seja permitida a definição de outra implementação em uma subclasse (propriedade Polimórfica igual a "*Final*"), ou que, ao contrário, permita a redefinição em uma subclasse (propriedade Polimórfica igual a "*Virtual*"). O nome da operação pode ser sobrecarregado, ou seja, em uma classe é possível ter várias operações com o mesmo nome, sendo a distinção entre as mesmas feita pelos tipos dos parâmetros esperados em cada operação.

Uma operação pode ter o direito de modificar o estado do objeto que a fornece (*"NonConst"*) ou deve preservar o estado do mesmo (*"Const"*). Uma operação pode ainda ser classificada, através de um estereótipo, em alguns tipos padronizados, tais como *"Constructor"*, *"Destructor"*, *"Read Accessor"*, *"Write Accessor"*, onde "*Constructor*" corresponde a uma operação que resulta na instanciação de um novo objeto da classe, "*Destructor*" corresponde à uma operação de destruição de um objeto da classe, "*Read Accessor*" corresponde ao retorno simples de um atributo pertencente à classe, e "*Write Accessor*" corresponde a uma simples atribuição efetuada em um atributo da classe. As operações de estereótipo "*Read Accessor*" e "*Write Accessor*" servem para esconder o formato de armazenamento dos atributos, permitindo o acesso aos mesmos apenas através destas operações. Estes dois últimos estereótipos podem ser utilizados na detecção de anti-padrões como, por exemplo, o *"Blob"*, descrito no capítulo 2. O estereótipo "*Constructor*", por sua vez, é utilizado na detecção de vários padrões e anti-padrões relacionados à instanciação de objetos.

Toda operação possui uma assinatura correspondente aos parâmetros esperados com os respectivos tipos, assim como o retorno esperado. Um parâmetro possui um nome, um tipo e uma direção, indicando se ele é recebido por valor, referência, ou se é um valor retornado pela operação.

3.2.6. Herança e Realização

A figura Figura 10 ilustra dois relacionamentos entre classificadores: *Herança* e *Realização*. *Herança* é um mecanismo através do qual elementos mais específicos podem incorporar a estrutura e o comportamento definido por elementos mais gerais. Isto vale tanto para classes (subclasses e superclasses) como para interfaces, onde uma interface mais específica pode ser definida pela expansão de uma interface mais genérica.

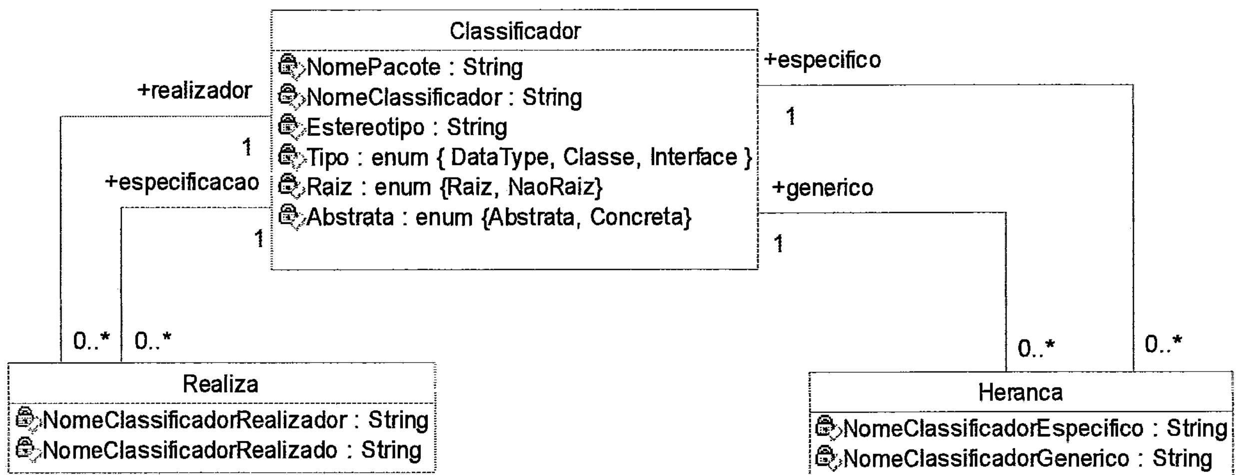


Figura 10: Relações entre classificadores

A *realização* de uma interface representa o relacionamento entre uma especificação e sua implementação. Uma especificação descreve o comportamento ou estrutura de algo sem determinar sua implementação. A semântica deste relacionamento indica que o realizador (uma classe) deve definir uma implementação para cada operação especificada pelo realizado (normalmente uma interface). Embora uma classe possua tanto a faceta de especificação como a de implementação, este trabalho restringe o relacionamento de realização, de modo que os elementos realizados serão sempre interfaces e os realizadores serão sempre classes, isto é, do mesmo modo como as

linguagens OO implementam este conceito. O relacionamento de realização é relativamente recente nas modelagens de projetos orientado a objetos, estando intimamente ligado a estruturas de implementação existentes em Java (interface) ou em mecanismos de desenvolvimento baseado em componentes tais como o COM (*Component Object Model*) e o CORBA (*Common Object Request Broker Architecture*) (SZYPERSKI, 1998).

3.2.7. Dinâmica dos objetos

Uma parte fundamental do metamodelo diz respeito à dinâmica de colaboração entre os objetos. A maioria dos trabalhos de detecção de padrões em projetos orientados a objetos limitam-se a trabalhar com um subconjunto dos elementos apresentados até aqui, ou seja, apenas elementos estruturais representados em diagramas de classes. Entretanto, a parte dinâmica é fundamental para a detecção de inúmeros padrões e anti-padrões de projeto (KELLER et al., 1999). O metamodelo proposto neste trabalho captura esta parte através da representação de quatro elementos dinâmicos fundamentais: instanciação de objetos, destruição de objetos, invocação de operação de objetos e acesso aos atributos dos objetos.

A figura 11 ilustra a primeira dinâmica fundamental, representando o processo de instanciação de objetos.

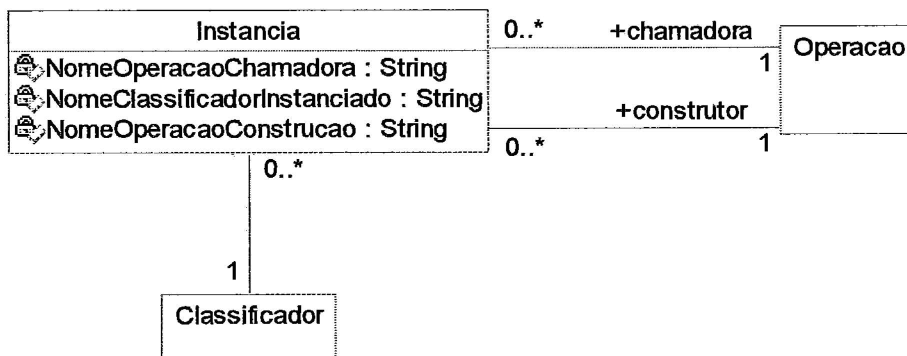


figura 11: Instanciação de um objeto

A entidade "Instancia" representa o momento de instanciação de um objeto correspondente a um classificador. Objetos são criados em tempo de execução como resultado da invocação de operações específicas de criação de objetos. Estas operações

são conhecidas como construtores, e a sua invocação indica uma dependência de uso entre o método que invoca esta criação ("*operação chamadora*") e a classe que será instanciada. Desta forma, podemos capturar todos os instantes de instanciação de um objeto, identificando exatamente o ponto em que isto acontece.

A figura 12 ilustra os elementos correspondentes à dinâmica de destruição de um objeto, representando o momento onde um objeto é destruído. A destruição de um objeto corresponde à invocação de uma operação com o estereótipo de destrutor. Como várias linguagens implementam destrutores virtuais, o classificador capturado por esta entidade pode não corresponder exatamente à classe do objeto que será destruído. Ele indica apenas qual a visibilidade, isto é, a dependência de tipo, que existe entre o método que está solicitando a destruição e o classificador.

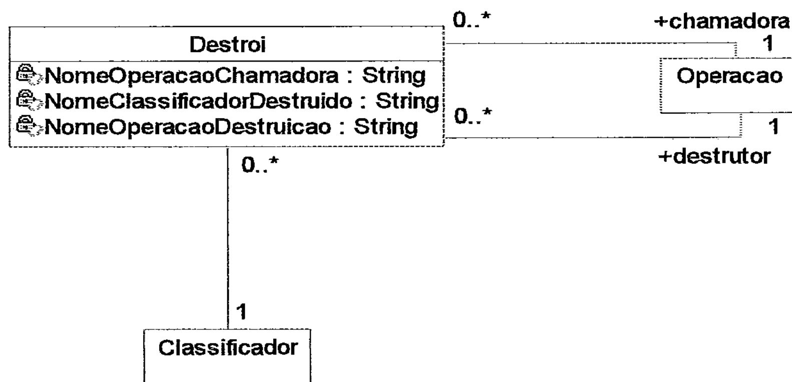


figura 12: Destruição de um objeto

A Figura 13 ilustra a terceira dinâmica capturada pelo metamodelo, correspondente à invocação de uma operação de um classificador.

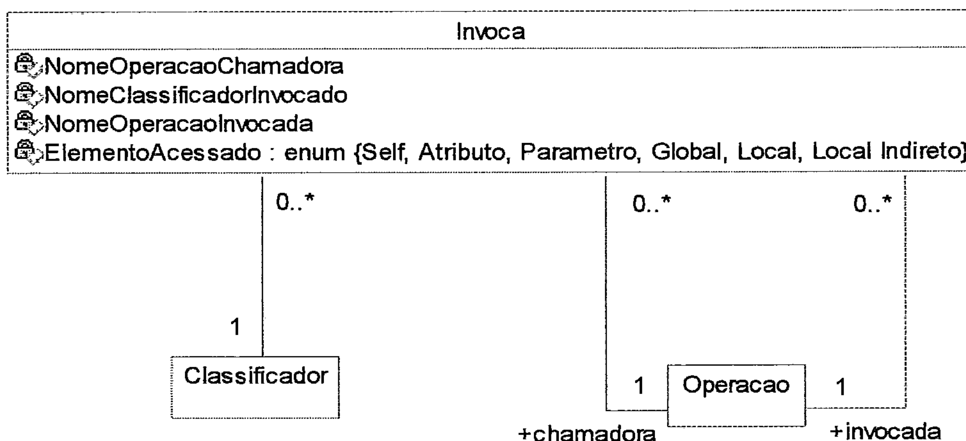


Figura 13: Invocação de operação

Através desta entidade, é possível capturar as interações entre os objetos. Uma chamada de operação é a invocação síncrona de um procedimento, onde o método chamador aguarda o término da operação invocada para voltar a ter o controle de execução. *NomeClassificadorInvocado* corresponde ao classificador que define a operação que está sendo invocada. Em função do polimorfismo, este atributo pode não corresponder exatamente à classe do objeto que será referenciado. Ele indica apenas qual a visibilidade, ou dependência de tipo, que existe entre o método solicitante e o classificador. *ElementoAcessado* indica como o método obteve acesso ao elemento cuja operação está sendo solicitada. Este atributo pode assumir os seguintes valores:

- "Self" (invocação de uma operação do próprio objeto);
- "Atributo" (invocação de uma operação de um elemento que é definido como um atributo ou pseudo-atributo do chamador);
- "Parametro" (o elemento foi recebido como parâmetro da operação chamadora);
- "Global" (o elemento está disponível globalmente para o sistema, através de uma variável global, por exemplo);
- "Local" (o elemento chamado foi criado localmente na operação chamadora);
- "Local Indireto" (o elemento chamado foi obtido através da invocação de uma operação de um outro objeto, normalmente uma chamada em cadeia, como, por exemplo, o elemento filme na invocação da operação obterPreco em: fita.obterFilme().obterPreco().

A Figura 14 ilustra o último elemento dinâmico capturado pelo metamodelo, que corresponde ao acesso a atributos de um objeto. Esta entidade captura o atributo acessado e o método responsável pelo acesso. Um acesso a um atributo pode ser a leitura, atualização, envio de mensagem, enfim qualquer referência ao atributo em um determinado método. A implementação dos métodos muitas vezes envolvem a manipulação de atributos do objeto ou da classe envolvida. Embora não seja recomendado, é possível implementar em muitas linguagens de programação o acesso direto a atributos de outros objetos da mesma classe ou até mesmo de outras classes.

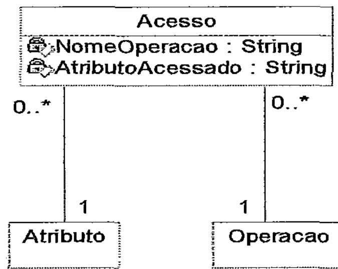


Figura 14: Acesso a um atributo de um objeto

A análise do acesso aos atributos de uma classe é útil na investigação de anti-padrões ligados à falta de coesão de uma classe, onde, por exemplo, exista dois ou mais subconjuntos disjuntos de métodos que manipulem subconjuntos também disjuntos de atributos da classe, indicando uma possível aglutinação indevida de duas abstrações em uma só.

3.3. Extrator de Projeto

Este componente tem como objetivo extrair informações de projeto a partir de um código fonte escrito em uma linguagem orientada a objetos como, por exemplo, C++, Java, Smalltalk, armazenando-as automaticamente em uma ferramenta CASE de modelagem de software orientado a objetos. O código fonte deve ser analisado com o objetivo de recuperar as informações de projeto, conforme o metamodelo descrito na seção 3.2. Os dados extraídos por este componente devem permitir uma posterior análise de sistemas legados, em busca de construções que possam comprometer a sua manutenção futura, uma vez que, na maior parte destes sistemas, a única fonte de informação para este tipo de análise é seu código fonte. Os modelos, quando existentes, são superficiais ou, muitas vezes, não refletem a realidade do código.

Embora as ferramentas CASE mais utilizadas no desenvolvimento de software na indústria, tais como o Rational Rose (RATIONAL, 1998), Paradigm Plus (PLATINUM, 1999), Select Enterprise (SELECT, 1999), disponham de módulos de engenharia reversa para linguagens como C++ e Java, as informações extraídas do código fonte limitam-se àquelas que podem ser obtidas com uma análise estrutural do mesmo. A engenharia reversa de código C++, por exemplo, recupera estas informações a partir da análise dos "*header files*", ou seja, apenas analisando a declaração da classe, seus atributos e operações. Desta forma, é possível somente recuperar informações

estruturais tais como pacotes, classes, atributos, operações, relações de herança, limitando o escopo de detecção de construções de projeto. Em função disto, identificamos a necessidade de agregar à ferramenta CASE um módulo de extração de informações de projeto que seja capaz de extrair não somente as construções estruturais, como também outras informações relativas à implementação dos métodos das classes do modelo, correspondentes à dinâmica de colaboração, instanciação e destruição de objetos. Desejamos obter as seguintes informações com esta extração:

- a estrutura de todos os classificadores, seus atributos e operações, a assinatura de cada operação (seus parâmetros e retorno), relacionamentos de herança e realização de interfaces, assim como o agrupamento destes classificadores em pacotes, de acordo com o metamodelo definido na seção 3.2. Este agrupamento em pacotes pode ser obtido a partir da organização dos arquivos fonte em diretórios, conforme descrito em (BOOCH et al., 1999) ou por definições explícitas existentes em linguagens como Java, onde a construção *"package"* indica diretamente o agrupamento dos elementos em pacotes.

- estereótipo dos métodos (*"constructor"*, *"destructor"*, *"read accessor"*, *"write accessor"*). Os estereótipos *"constructor"* e *"destructor"* podem ser diretamente extraídos do código. Os estereótipos *"accessor"* correspondem a métodos que retornam ou fazem uma atribuição simples de valor a um atributo da classe. O código da Figura 15 mostra um exemplo de código em C++ com métodos correspondentes a estes estereótipos.

```
class Fita {
private:
      string   numeroFita;
public:
      // método construtor (mesmo nome da classe)
      Fita();
      // método destrutor (mesmo nome da classe, precedido por um ~)
      ~Fita();
      // read accessor - atributo numeroFita
      string getNumero() { return numeroFita; }
      // write accessor - atributo numeroFita
        void setNumero(string s) { numeroFita = s; }
};
```

Figura 15: Exemplo de métodos e seus estereótipos

- os atributos manipulados em cada método das classes do projeto;

- as colaborações necessárias para a implementação de cada método, através da obtenção de todas as invocações de operações, instanciações e destruições de objetos. Esta análise de colaborações é feita com o objetivo de obter dependências entre tipos, não sendo necessário saber exatamente a classe do objeto cuja operação foi invocada, como em casos onde a invocação é realizada de maneira polimórfica. A obtenção de informações sobre dependências entre tipos é suficiente para predizer implicações de modificações no projeto, avaliar a qualidade do projeto e sugerir melhorias ou ainda para construir visões de mais alto nível de grandes sistemas. (CIUPKE, 1998).

No caso específico das colaborações existentes na implementação de um método, o extrator deve recuperar, para cada método, as invocações, instanciações e destruições de objetos realizadas diretamente pelo método, ou seja, apenas em um nível de chamada. A Figura 16 ilustra um exemplo desta extração:

```
class EmprestimoFita {
public:
   void acrescentarFita(Fita aFita) {..; aFita.preco(); .. }
};

class Fita {
private:
   Filme   oFilme;
public:
   currency preco() {...; oFilme.preco(); ... }
};

class Filme {
public:
   void preco() { ...; }
};
```

Figura 16: Exemplo de extração de informações sobre troca de mensagens.

Neste exemplo, teríamos a extração de informações sobre o método *acrescentarFita,* indicando que ele invoca a operação *preco* de um objeto da classe *Fita* ou descendente de *Fita,* ou seja, existe uma dependência da classe *EmprestimoFita* em relação ao tipo *Fita.* Por outro lado, a informação sobre a chamada da operação *preco* da classe *Filme* seria atrelada apenas ao método *preco* da classe *Fita,* isto é, apenas ao método diretamente chamador. Desta forma, extraímos de cada método as colaborações diretamente disparadas a partir do mesmo.

Para cada invocação de operação identificada em um determinado método, devemos capturar as seguintes informações: a operação chamada, o tipo (classificador) do objeto referenciado pelo método, e como este objeto está sendo acessado pelo método (parâmetro, objeto criado localmente, objeto acessado globalmente, atributo da classe, ou o próprio objeto). No exemplo anterior, teríamos que na colaboração *aFita.preco(),* presente no método *acrescentarFita* da classe *Emprestimo,* o objeto colaborador *(aFita)* é um parâmetro do método, enquanto que na colaboração *oFilme.preco,* existente no método *preco* da classe *Fita,* o objeto colaborador *(oFilme)* é um atributo do próprio objeto invocador.

3.4. Gerador de Fatos

Uma vez que as informações de projeto estejam disponíveis na ferramenta CASE, seja através da extração realizada pelo componente *Extrator de Projeto* (em um processo de engenharia reversa), seja através da própria construção do modelo pelo projetista (no caso de um processo de modelagem de um novo sistema), o componente *Gerador de Fatos* é responsável pela geração de uma base de fatos com os elementos capturados do modelo de projeto armazenado pela ferramenta CASE.

O objetivo de transcrever os elementos de projeto existentes na ferramenta CASE para uma base de fatos é obter uma representação do projeto de forma independente da ferramenta CASE utilizada para modelagem, de modo a permitir uma posterior manipulação destes elementos através, por exemplo, de consultas ou de buscas por determinadas construções (padrões e anti-padrões). A geração da base de fatos é realizada a partir das informações presentes nos diagramas de classes, nos diagramas de colaboração que capturam os aspectos dinâmicos no contexto de cada operação, e nas especificações de cada elemento semântico representado nestes diagramas.

Uma possível forma de representação dos fatos é através de predicados em Prolog. A Figura 17 ilustra uma lista de predicados que podem ser utilizados para expressar as construções básicas de projetos orientado a objetos. Estes predicados são definidos em conformidade com as entidades do metamodelo descrito na seção 3.2. No apêndice I, estes predicados e todos os seus termos são descritos detalhadamente.

- Pacote (NomePacote, Estereótipo, NomePacotePai)
- DependenciaPacote (NomePacoteCliente, NomePacoteFornecedor)
- Classificador (NomePacote, NomeClassificador, Estereótipo, Tipo, Abstrata, Folha, Raiz)
- VisibilidadePacote (NomePacote, NomeClassificador, Visibilidade)
- Realiza (NomeClassificadorRealizador, NomeClassificadorRealizado)
- HerdaDe (NomeClassificadorEspecifico, NomeClassificadorGenerico)
- Atributo (NomeClassificador, NomeAtributo, Escopo, Visibilidade, TipoAtributo, ClassificadorAssociado, Modificabilidade, Multiplicidade, Agregação)
- Operacao (NomeClassificador, NomeOperação, Escopo, Visibilidade, Estereótipo, Polimórfica, Abstrata, Const)
- Parametro (NomeOperação, NomeParâmetro, Ordem, Direção, TipoParâmetro)
- Invoca (NomeOperaçãoChamadora, NomeClassificadorInvocado, NomeOperaçãoInvocada, ElementoAcessado)
- Instancia (NomeOperaçãoChamadora, NomeClassificadorInstanciado, NomeOperaçãoConstrução)
- Destroi (NomeOperaçãoChamadora, NomeClassificadorDestruído, NomeOperaçãoDestruição)
- Acesso (NomeOperação, AtributoAcessado)

Figura 17: Predicados utilizados para expressar fatos sobre um projeto OO

3.5. Captura de Conhecimento

Este componente é responsável pela captura do conhecimento disponível na literatura e através de especialistas em projeto orientado a objetos. Este conhecimento é capturado na forma de heurísticas, padrões e anti-padrões. O resultado desejado é deixar à disposição do projetista uma base de conhecimento que descreva, de forma organizada e padronizada, os princípios de um bom projeto orientado a objetos, as construções que contribuam para que o projeto esteja em conformidade com estes princípios, bem como as construções que resultem em possíveis violações destes princípios. Este conhecimento deve ser capturado de forma a tornar explícitas as relações existentes entre as heurísticas, os padrões e anti-padrões catalogados.

As várias heurísticas disponíveis na literatura estão descritas de forma muitas vezes implícita, não existindo um padrão de descrição das mesmas. Este trabalho propõe que as heurísticas, independente de sua origem, sejam catalogadas utilizando um formato padronizado. As informações básicas que devem ser capturadas sobre uma heurística são as seguintes:

- **Título**: frase breve que descreva a essência da heurística.
- **Descrição:** detalhamento sobre o que o projetista deve fazer para manter seu projeto em conformidade com a heurística.

- **Motivação:** razões que explicam porque esta heurística possibilita a geração de melhores projetos, e que, portanto, motivam o projetista a segui-la.

- **Fonte:** origem ou autor da heurística.

Além destas informações, o projetista deve ter acesso, a partir de uma heurística, às possíveis formas de violação das mesmas, capturadas como anti-padrões. Desta forma, dada uma heurística, deve ser possível visualizar e associar possíveis anti-padrões relacionados. De forma análoga, o projetista deve poder visualizar e associar construções típicas de projeto (padrões) que contribuam para que esta heurística seja seguida.

Assim como as heurísticas, os padrões de projeto também devem ser catalogados seguindo um formato padronizado para sua descrição. Este formato é baseado no formato *GoF* de descrição de padrões (GAMMA et al., 1995), descrito na seção 2.3, onde a definição de um padrão é composta pelas seções: Nome, Problema, Contexto, Solução, Consequências, Padrões Relacionados e Uso Conhecido. Além disso, propõe-se o acréscimo de mais três seções ao formato *GoF*:

- **Fonte:** origem ou autor do padrão, uma vez que existem vários catálogos de padrões na literatura, e a própria organização pode desenvolver os seus próprios padrões.

- **Anti-Padrões Relacionados**: aqui deverão estar relacionadas possíveis construções correspondentes a soluções problemáticas, tipicamente aplicadas por quem não conheça o padrão. O objetivo é possibilitar a indicação de oportunidades para a aplicação de um padrão, assim como disponibilizar uma fonte de referência que auxilie o projetista a evitar estas soluções tecnicamente piores.

- **Detecção:** indicação de como o padrão pode ser reconhecido no projeto, ou seja, as características que identificam este padrão. Mais adiante nesta seção, este tópico é descrito com mais detalhes.

O terceiro tipo de conhecimento a ser capturado por este componentes corresponde aos anti-padrões. Da mesma forma que as heurísticas e os padrões, os anti-

padrões também devem ser capturados seguindo um formato padronizado dividido em seções, conforme descrito na seção 2.4. A este formato, propõe-se a inclusão das seguintes seções:

- **Fonte:** origem ou autor do anti-padrão.

- **Heurísticas Violadas:** aqui devem estar relacionadas as heurísticas de projeto que serão violadas com esta construção, caracterizando-a como um anti-padrão.

- **Detecção:** indicação de como o anti-padrão pode ser reconhecido no projeto, ou seja, as características que identificam este anti-padrão.

- **Solução:** esta seção já existe no padrão proposto por (BROWN et al., 1998), sendo que foi acrescentada uma relação com padrões de projeto, uma vez que a solução de um anti-padrão pode ser a sua substituição por um padrão já catalogado. Exemplo: um objeto declarado globalmente constitui um anti-padrão de projeto que pode ser substituído pela aplicação do padrão de projeto *"Singleton"* (GAMMA et al., 1995).

Este componente deve permitir a administração do repositório de heurísticas, padrões e anti-padrões e a sua evolução, ou seja, a inclusão de novos elementos, modificação de elementos existentes e remoção de elementos não mais utilizados. Além disso, deve prover um mecanismo de organização e classificação dos elementos, isto é, o usuário deve poder organizar as heurísticas, padrões e anti-padrões em categorias de acordo com a sua conveniência.

Este componente é também responsável pela definição das informações que possibilitem a posterior detecção de padrões e anti-padrões em um projeto orientado a objetos. Esta definição corresponde à seção *Detecção* adicionada ao formato de definição dos padrões e anti-padrões, conforme descrito anteriormente. A partir dos padrões, das heurísticas e dos anti-padrões disponíveis na literatura, e também do conhecimento resultante da experiência de especialistas em projetos OO, deve-se capturar este conhecimento e formalizá-lo em regras que definam as características necessárias para o reconhecimento de uma determinada construção boa ou ruim em um projeto orientado a objetos.

Uma vez que cada padrão ou anti-padrão pode ser capturado a partir de regras, novos padrões ou anti-padrões podem ser detectados a partir do momento em que suas respectivas regras tenham sido definidas, o que possibilita a evolução deste conhecimento como resultado da própria experiência da organização no desenvolvimento e manutenção de sistemas orientados a objetos. Uma possível forma de representação destas regras é através da utilização dos mesmos predicados Prolog utilizados para a representação de projetos orientado a objetos, conforme descrito na seção 3.4.

A Figura 18 apresenta um exemplo com a formalização do padrão "*AbstractFactory*" (GAMMA et al., 1995) utilizando Prolog. Além de permitir a detecção deste padrão, é possível também capturar quais são as classes que preenchem os papéis correspondentes aos participantes deste padrão. Neste exemplo, estes papéis correspondem ao de *AbstractFactory*, as várias *ConcreteFactories*, os *AbstractProducts* e suas respectivas subclasses *ConcreteProducts*.

```
abstractFactoryPattern(AbstractFactory,ConcreteFactories,AbstractProducts,ConcreteProducts)
:-      abstractFactory(AbstractFactory,AbstractProducts),
        findAll(ConcreteFactory,concreteFactory(ConcreteFactory,AbstractFactory),
        ConcreteFactories), listSize(ConcreteFactories, NumFact), NumFact > 0,
        findAll (X, concreteProduct (X, AbstractProducts), ConcreteProducts).
```

Figura 18: padrão "*Abstract Factory*"

A regra ilustrada na Figura 18 define as condições para a detecção do padrão "*AbstractFactory*". Esta regra é definida em função de predicados auxiliares que são listados da Figura 19 até a Figura 25.

```
abstractFactory (Classe, AbstractProducts) :-
  classificador(_, Classe, _, "Classe", "Abstrata", "NaoFolha", "Raiz"),
  findAll(AbstractProduct,returnsAbstractProduct(Classe, AbstractProduct),
  AbstractProducts), listSize(AbstractProducts, Tam), Tam > 0.
```

Figura 19: Detecção de um participante no papel de *AbstractFactory*

```
concreteFactory (Classe, AbstractFactory) :-
        classificador (_, Classe, _, "Classe", "Concreta", _, _),
        descendente(Classe, AbstractFactory),
        abstractFactory(AbstractFactory, AbstractProducts), findAll(ConcreteProduct,
        instantiatesConcreteProduct (Classe, ConcreteProduct, AbstractProducts),
        ConcreteProducts), listSize(ConcreteProducts, Tam), Tam > 0.
```

Figura 20: Detecção de um participante no papel de *ConcreteFactory* descendente de um *AbstractFactory*

```
abstractProduct (Product) :-
        classificador(_,Product,_,"Classe", "Abstrata","NaoFolha","Raiz"),
        classificador(_,ConcreteProduct, _, "Classe","Concreta",_,_),
        ancestral (Product, ConcreteProduct).
```

Figura 21: Detecção de um participante no papel de *AbstractProduct*

```
concreteProduct (ConcreteProduct, AbstractProducts) :-
        descendente (ConcreteProduct, X),
        member (X, AbstractProducts).
```

Figura 22: Detecção de um participante no papel de *ConcreteProduct*, descendente de um *AbstractProduct*.

```
returnsAbstractProduct (Classe, AbstractProduct) :-
        operacao(Classe, Oper, _, "Instancia", "Public", _, "Virtual", _, _),
        parametro(Oper, _, _, "Return", AbstractProduct),
        abstractProduct (AbstractProduct).
```

Figura 23: Verifica se um classificador possui uma operação que retorna um *AbstractProduct*.

```
instantiatesConcreteProduct (Classe, ConcreteProduct, AbstractProducts) :-
        operacao(Classe, Oper, _, "Instancia", "Public", _, "Virtual", _, _),
        parametro(Oper, _, _, "Return", AbstractProduct),
        member (AbstractProduct, AbstractProducts),
        instancia(Oper, ConcreteProduct, _),
        Classe <> ConcreteProduct,
        descendente(ConcreteProduct, AbstractProduct).
```

Figura 24: Verifica se um classificador possui uma operação que instancia um *ConcreteProduct*.

```
descendente (X, Y)     :- herdade(X, Y).
descendente (X, Y)     :- herdade(Z, Y), descendente(X, Z).
ancestral  (X, Y)      :- herdade(Y, X).
ancestral (X, Y)       :- herdade(Z, X), ancestral(Z, Y).
```

Figura 25: Identifica relações de herança direta ou indireta entre classificadores.

Da mesma forma, é possível catalogar os anti-padrões com regras estabelecidas a partir dos mesmos predicados que representam as construções OO básicas definidas no metamodelo. A seguir, são apresentados alguns exemplos de anti-padrões formalizados utilizando Prolog.

O padrão *"PublicVisibility"* (Figura 26) detecta a definição de atributos na área pública de uma classe. Este fato vai contra a heurística de projeto *"Todos os dados*

devem estar escondidos na sua classe" (RIEL, 1996).

```
publicVisibility (Classe, Atributo) :-
        classificador(_, Classe, _, _, _, _, _),
        atributo(Classe, Atributo, _, "Public", _, _, _, _, _).
```

Figura 26: Anti-Padrão "*PublicVisibility*"

O anti-padrão "*ProtectedVisibility*" (Figura 27) detecta a definição de atributos na área *protected* de uma classe. Este fato vai contra a heurística de projeto "*Todos os dados de uma classe base devem ser privados*" (RIEL, 1996).

```
protectedVisibility(Classe, Atributo) :-
        classificador(_, Classe, _, _, _, _, _),
        atributo(Classe, Atributo, _, "Protected", _, _, _, _, _).
```

Figura 27: Anti-Padrão "*ProtectedVisibility*"

O anti-padrão "*ExpositionOfAuxiliaryMethod*" (Figura 28) detecta a definição de métodos na interface pública da classe que servem apenas para a implementação de outras operações da própria classe. Este fato vai contra a heurística de projeto "*Não coloque detalhes de implementação na interface pública de uma classe*" (RIEL, 1996).

```
expositionOfAuxiliaryMethod (Classe, Metodo) :-
        classificador(_, Classe, _, _, _, _, _),
        operacao(Classe, Metodo, _, _, "Public", _, _, _, _),
        findAll(ClasseCliente,externalClient(Metodo,ClasseCliente),ClassesClientes),
        listSize(ClassesClientes, NumExternos), NumExternos = 0,
        findAll(ClasseCliente,internalClient(Metodo,ClasseCliente,ClientesInternos),
        listSize(ClientesInternos, NumInternos), NumInternos > 0.

externalClient (Metodo, ClasseCliente) :-
        operacao (Classe, Metodo, _, _, _, _, _, _, _),
        operacao (ClasseCliente, Chamador, _, _, _, _, _, _, _),
        invoca (Chamador, Classe, Metodo, _),
        not (mesmaHierarquia(ClasseCliente, Classe)).

internalClient (Metodo, ClasseCliente) :-
        operacao (Classe, Metodo, _, _, _, _, _, _, _),
        operacao (ClasseCliente, Chamador, _, _, _, _, _, _, _),
        invoca (Chamador, Classe, Metodo, _),
        mesmaHierarquia(ClasseCliente, Classe).

mesmaHierarquia (X, Y) :- X = Y.
mesmaHierarquia (X, Y) :- descendente (X, Y).
mesmaHierarquia (X, Y) :- ancestral (X, Y).
```

Figura 28: Anti-Padrão "*ExpositionOfAuxiliaryMethod*"

Além de anti-padrões resultantes de violações de heurísticas de projeto OO, podemos também identificar outras construções potencialmente inadequadas através de uma análise dos problemas solucionados pelos padrões de projeto. Como exemplo desta segunda estratégia, apresentamos dois anti-padrões associados a alguns padrões de instanciação de objetos (GAMMA et al., 1995).

As duas regras definidas na Figura 29 detectam a presença de um objeto com

escopo global, cuja classe poderia ser melhor definida através da aplicação do padrão de projeto *"Singleton"*. A primeira regra corresponde à definição de uma instância com escopo global, enquanto que a segunda corresponde à definição, em um classificador, de um atributo com visibilidade pública e escopo de classe.

```
globalScopeObject (Objeto, Classe) :-
        classificador (_, "LogicalView::Global", _, _, _, _, _),
        atributo ("LogicalView::Global", Objeto, _, "Public", _, Classe, _, _, _).

globalScopeObject (Objeto, Classe) :-
        classificador (_, X, _, _, _, _, _),
        atributo (X, Objeto, "Classe", "Public", _, Classe, _, _, _).
```

Figura 29: anti-padrão "*Global Scope Object*"

O anti-padrão *"DisperseInstantiation"* (Figura 30) reflete o acoplamento de vários pontos do projeto a uma determinada classe e, portanto, a uma determinada implementação. Este acoplamento se caracteriza pela instanciação direta desta classe em várias outras classes do projeto. O projeto ficaria mais flexível se um padrão como, por exemplo, *"AbstractFactory"* ou *"Prototype"*, fosse adotado no lugar da instanciação direta em vários pontos do software.

```
disperseInstantiation (Classe) :-
        findAll (Creator, doInstantiation(Creator, Classe), Creators),
        listSize (Creators, Tam),
        Tam > 1).

doInstantiation (Creator, Classe) :-
        operacao (Creator, Oper, _, _, _, _, _, _, _),
        instancia (Oper, Classe, _).
```

Figura 30: anti-padrão *"DisperseInstantiation"*

3.6. Analisador

Este componente é responsável por analisar a base de fatos correspondente ao projeto orientado a objetos que está sendo objeto de verificação em busca de construções catalogadas na base de conhecimento pelo componente *Captura de Conhecimento*, descrito na seção 3.5.

Os padrões e anti-padrões estão representados na forma de regras, conforme descrito na seção 3.5, enquanto que o projeto analisado está representado em uma base

de fatos. Este componente é responsável por verificar quais regras, de acordo com uma seleção feita pelo projetista, são satisfeitas por esta base de fatos. O projetista pode selecionar, de diversos modos, as construções que ele deseja verificar em um projeto. A seguir, estão relacionadas algumas facilidades de seleção disponibilizadas por este componente:

- seleção de uma categoria de heurísticas, ou de uma heurística específica, obtendo-se, como resultado, possíveis violações, correspondentes aos anti-padrões relacionados com as heurísticas selecionadas.

- seleção de um padrão de projeto, ou de um conjunto de padrões, obtendo-se, como resultado, possíveis construções que poderiam ser melhoradas através da aplicação destes padrões, ou seja, oportunidades para a aplicação de um padrão ou conjunto de padrões.

- seleção direta de um anti-padrão de projeto, ou de um conjunto de anti-padrões, obtendo-se, como resultado, as ocorrências deste anti-padrão no projeto. O objetivo é permitir ao projetista identificar pontos no projeto onde ocorram construções que possam comprometer a futura manutenção do sistema. Além de identificar estes pontos, ele deve poder extrair informações da base de conhecimento sobre como estas construções poderiam ser reestruturadas, sendo que esta reestruturação muitas vezes pode corresponder à aplicação de um padrão de projeto.

- seleção de um fragmento do projeto (um classificador ou conjunto de classificadores) e verificação das heurísticas que eventualmente foram violadas neste fragmento, assim como a detecção da participação de elementos deste fragmento em algum padrão, ou anti-padrão, de projeto.

- ou ainda, seleção de um padrão, ou de conjunto de padrões, obtendo-se, como resultado, os pontos do projeto onde estes padrões ocorram, além do enquadramento dos elementos do projeto como participantes em cada padrão.

No caso de ser encontrado, por exemplo, o anti-padrão "*PublicVisibility*", deve ser possível visualizar o atributo e a classe em que isto ocorreu, indicando como uma possível solução, a colocação deste atributo na área *private* da classe, e a criação de

métodos de acesso *(get, set)* para a recuperação e modificação desta informação.

Se, por exemplo, uma instância do padrão *"Abstract Factory"* for detectada, deve ser possível indicar não somente a presença deste padrão no projeto, mas também as classes correspondentes aos papéis de *"Abstract Factory"*, *"Concrete Factories"*, *"Abstract Products"* e *"Concrete Products"* nesta instância do padrão.

3.7. Conclusões

Este capítulo apresentou uma arquitetura de apoio para a análise de modelos orientados a objetos através da integração de heurísticas, padrões e anti-padrões a ferramentas CASE de projeto orientado a objetos. Com esta integração, o projetista passa a ter à sua disposição informações importantes para a produção de projetos flexíveis e reutilizáveis. Estas informações devem ser organizadas de modo a deixar explícito os relacionamentos existentes entre elas. Com isto, o projetista pode, por exemplo, visualizar heurísticas para construção de um bom projeto orientado a objetos e, ao mesmo tempo, saber que construções típicas constituem violações destas heurísticas, devendo, portanto, ser evitadas.

Além de disponibilizar uma base de conhecimento para o projetista, esta arquitetura também objetiva fornecer suporte para a detecção de construções típicas em um projeto orientado a objetos. Estas construções correspondem a padrões e anti-padrões catalogados nesta base. Esta detecção é baseada em informações estruturais do projeto, isto é, aquelas disponíveis em diagramas de classes e, também, ao contrário de trabalhos como os descritos no capítulo 2, em informações sobre a dinâmica de colaboração entre objetos. Estas informações são capturadas conforme um metamodelo que contempla os elementos de um projeto orientado a objetos em um nível de detalhe suficiente para a detecção de padrões tanto estruturais como comportamentais.

Os elementos de um projeto a ser analisado são extraídos a partir da ferramenta CASE, gerando uma base de fatos que serve de matéria-prima para uma análise, guiada pelo projetista, em busca de padrões ou de anti-padrões catalogados na base de conhecimento. Embora a ênfase das pesquisas nesta área ainda seja na detecção de padrões, conforme ilustrado pelos trabalhos descritos no capítulo 2, a detecção de anti-

padrões de forma integrada com heurísticas e padrões possibilita aos projetistas, especialmente os menos experientes, uma forma de se beneficiarem de um conhecimento, que eles ainda não possuem, sobre construções de projeto que devem ser evitadas. A possibilidade de detectar estas construções em tempo de projeto pode evitar a propagação das mesmas no código correspondente. Entretanto, se estas construções estiverem presentes em sistemas já implementados, esta detecção possibilita identificar pontos nestes sistemas que devam ser reestruturados, indicando ainda, um caminho para esta reestruturação, muitas vezes, através da aplicação de um padrão de projeto.

Esta atividade de detecção, seja em um modelo de projeto ou em código fonte, é trabalhosa e sujeita a erros ou esquecimentos, considerando o grande volume de heurísticas, padrões e anti-padrões hoje existentes. Desta forma, um suporte automatizado para esta detecção passa a ter especial importância. No próximo capítulo, é apresentada a implementação de um protótipo desta proposta.

Capítulo 4

OOPDTool – Uma Ferramenta de Apoio à Detecção de Padrões e Anti-Padrões

4.1. Introdução

O capítulo 3 descreveu a proposta de uma arquitetura de apoio para a análise de modelos orientados a objetos baseada na integração do conhecimento sobre heurísticas, padrões e anti-padrões a ferramentas CASE de projetos orientados a objetos. Este capítulo apresenta o protótipo de uma ferramenta correspondente à implementação de parte desta proposta, denominada OOPDTool. Esta implementação concentrou-se em dois aspectos principais da arquitetura:

- a catalogação de heurísticas, padrões, anti-padrões e de seus relacionamentos.
- a detecção de padrões e anti-padrões em um projeto orientado a objetos.

A arquitetura proposta e a implementação do protótipo foram utilizados em um experimento prático a partir de produtos gerados no contexto do projeto Odyssey (BRAGA et al., 1998), em desenvolvimento na COPPE/UFRJ. Odyssey é um projeto de pesquisa que visa a construção de uma infra-estrutura de reutilização baseada em modelos de domínio, englobando um conjunto de ferramentas para aquisição de conhecimento sobre o domínio, gerência de componentes, busca de informações sobre o domínio baseada em agentes inteligentes, transformação de modelos conceituais em componentes arquiteturais, além de ferramentas para a especificação e projeto de aplicações do domínio utilizando a tecnologia de orientação a objetos.

Duas das ferramentas deste ambiente foram analisadas neste experimento: PAC - ferramenta para configuração de processos de aquisição de conhecimento (ROSETI et al., 1998) e o editor de diagramas, destinado à elaboração dos modelos definidos em (BRAGA e WERNER, 1999), cuja notação é baseada na UML (OMG, 1998). As duas ferramentas foram implementadas em Java, sendo que a versão analisada do editor de

diagramas englobava apenas o suporte para a elaboração de diagramas de casos de uso e de classes, embora uma versão mais completa, incluindo outros tipos de diagramas ("*features*", sequência e transição de estados) tenha sido posteriormente construída, absorvendo, inclusive, os resultados obtidos com este experimento.

Este capítulo está organizado da seguinte forma: a seção 4.2 apresenta aspectos de integração do protótipo com a ferramenta CASE de modelagem de projeto utilizada. A seção 4.3 apresenta uma visão geral do protótipo. A implementação dos componentes *Captura de Conhecimento, Geração de Fatos e Analisador* é apresentada nas seções 4.4, 4.5 e 4.6, respectivamente. A seção 4.7 apresenta um estudo de caso realizado no contexto do projeto Odyssey. Finalizando o capítulo, a seção 4.8 apresenta algumas conclusões sobre a utilização prática da abordagem proposta.

4.2. Interoperabilidade com a ferramenta CASE

A ferramenta CASE, de acordo com a arquitetura descrita no capítulo 3, é responsável por manter todas as informações do projeto, sendo que elas podem ser definidas pelo projetista diretamente na ferramenta, ou recuperadas pelo componente *Extrator de Projeto*, através de um processo de engenharia reversa. Estas informações disponibilizadas pela ferramenta CASE constituem a matéria-prima para a detecção dos padrões e anti-padrões realizada pela OOPDTool.

Embora a arquitetura proposta no capítulo 3 não seja específica para uma ferramenta CASE em particular, a funcionalidade disponibilizada pela implementação deste protótipo, especificamente a detecção de padrões e anti-padrões, foi limitada aos projetos realizados com o apoio da ferramenta CASE Rose 98 (RATIONAL, 1998). Esta ferramenta fornece suporte às atividades de análise e projeto orientado a objetos, podendo gerar modelos segundo as notações de Booch, OMT e UML. Além de ter sido a ferramenta CASE utilizada para apoiar o desenvolvimento das primeiras ferramentas do projeto Odyssey, um outro fator que motivou sua escolha foi o fato de sua arquitetura possibilitar a extensão das suas funcionalidades de diferentes maneiras, como, por exemplo:

- automação de funções através de "*scripts*" escritos em *Rose Scripting Language* (linguagem de programação nativa do Rose, baseada na linguagem

Summit BaseScript), permitindo criar elementos e diagramas, gerar documentos, entre outras facilidades.

- interoperabilidade com outras ferramentas, uma vez que as informações de um modelo de projeto do Rose 98 podem ser acessadas e modificadas por outras aplicações através de componentes OLE (*Object Linking and Embedding*) (BROCKSCHMIDT, 1995) por ela disponibilizados.

- customização dos seus menus, permitindo a execução de um programa externo, de um "*script*", ou ainda, a invocação de operações disponíveis em outros programas via automação OLE.

A Figura 31 ilustra os aspectos de interoperabilidade disponíveis no Rose 98. Através de uma interface de extensão, denominada REI (*Rose Extensibility Interface*), é possível acessar os seus três componentes principais (RATIONAL, 1998).

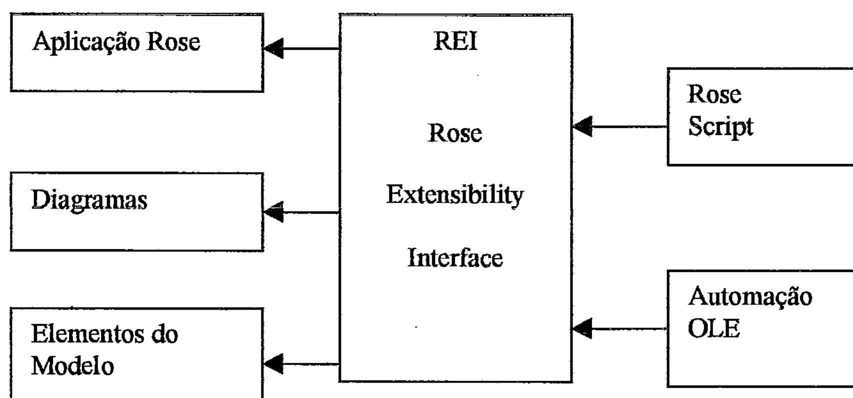


Figura 31: Aspectos de interoperabilidade do Rose98

- *Aplicação Rose*: objetos que permitem acesso à funcionalidade da aplicação Rose, tais como, criar um novo projeto, salvar o projeto corrente, executar um "*script*", entre várias outras funções. Através da elaboração de programas utilizando estes objetos, é possível realizar as mesmas operações disponíveis nos menus da ferramenta

- *Diagramas*: objetos que permitem obter informações de diagramas e elementos gráficos de um projeto aberto no Rose, além de possibilitar a sua

atualização.

- *Elementos do Modelo*: objetos que permitem o acesso às informações semânticas do modelo de projeto corrente no Rose, ou seja, informações sobre os pacotes, classes, atributos, enfim, sobre todos os componentes de um modelo orientado a objetos. No Rose, existe uma distinção clara entre a parte semântica do modelo (Elementos do Modelo) e as possíveis representações visuais dos mesmos (Diagramas).

O acesso a estes três componentes pode ser realizado de duas formas: através de "*scripts*" escritos em "*Rose Scripting Language*", ou através de automação OLE, a partir de uma aplicação escrita em qualquer ambiente que suporte este tipo de automação, como, por exemplo, Visual Basic, Delphi, Visual C++, Borland C++ Builder, entre outros.

4.3. OOPDTool – Visão Geral

Conforme descrito no capítulo 3, a arquitetura proposta abrange os seguintes componentes: o *Extrator de Projeto*, responsável pela captura de informações de projeto a partir de código fonte escrito em uma linguagem orientada a objetos; o *Gerador de Fatos*, responsável pela geração de uma base de fatos a partir das informações de projeto armazenadas na ferramenta CASE; o componente de *Captura de Conhecimento*, responsável pelo gerenciamento da base de conhecimento sobre projeto orientado a objetos na forma de heurísticas, padrões e anti-padrões, sendo responsável, também, pela definição das regras para detecção de padrões e anti-padrões em um projeto; e, por fim, o *Analisador*, responsável por analisar a presença de padrões e anti-padrões catalogados em um determinado projeto. A Figura 32 ilustra os principais elementos envolvidos na implementação deste protótipo.

O pré-requisito para a detecção de padrões e anti-padrões é a existência das informações sobre o projeto na ferramenta CASE, não importando se as mesmas sejam capturadas via *Extrator*, ou manualmente colocadas pelo projetista. Considerando que parte das funcionalidades necessárias para a obtenção destas informações está disponível na ferramenta Rose 98, através de um módulo de engenharia reversa a partir de código Java, o componente *Extrator de Projeto*, responsável pela extração de

informações de projeto orientado a objetos a partir do código fonte, não foi implementado. Desta forma, limitamos o escopo desta implementação, assumindo a existência deste módulo. No estudo de caso, mostraremos como este módulo de engenharia reversa de código Java disponível no Rose foi aproveitado para extrair parte das informações necessárias à detecção de padrões e anti-padrões.

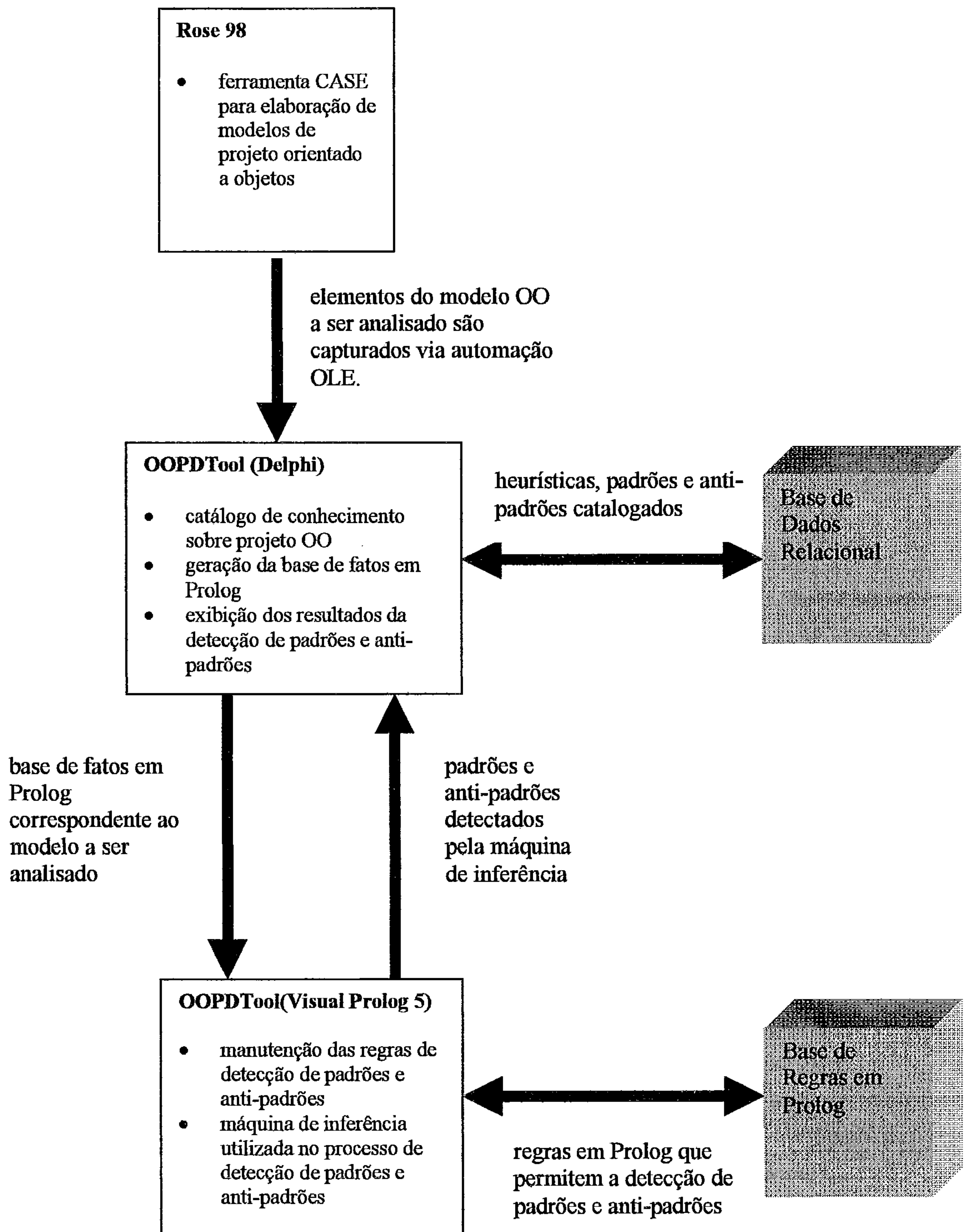


Figura 32: Principais elementos da implementação do OOPDTool

O componente *Gerador de Fatos*, implementado em Delphi 4, acessa as informações do projeto corrente no Rose via automação OLE, gerando, como resultado, um arquivo com os fatos correspondentes a estas informações. Estes fatos são descritos em Prolog utilizando os predicados descritos no apêndice I.

O componente de *Captura de Conhecimento* foi também implementado em Delphi 4. As informações sobre as heurísticas, padrões e anti-padrões são armazenadas em tabelas relacionais acessadas via BDE (*Borland Database Engine*), permitindo a utilização de qualquer base de dados relacional suportada pelo BDE. A manutenção das regras para detecção dos padrões e anti-padrões é realizada através do Visual Prolog 5.

O componente *Analisador* foi implementado em Delphi 4, utilizando os recursos da máquina de inferência do Visual Prolog 5. Basicamente, a implementação realizada em Delphi é responsável pela entrada de parâmetros para a detecção e pela exteriorização dos resultados, enquanto que a detecção propriamente dita dos padrões e anti-padrões é realizada pelo Visual Prolog, utilizando a base de fatos gerada e as regras para detecção destas construções.

A Figura 33 ilustra um modelo de classes utillizado na implementação do módulo OOPDTool – Delphi, responsável pela captura do conhecimento e por toda a a interação com o usuário na detecção dos padrões e anti-padrões. Heurísticas, padrões e anti-padrões são itens de conhecimento gerenciados pela ferramenta. Cada heurística está associada a uma categoria. Cada categoria pode ser composta por várias categorias subordinadas (sub-categorias) em uma estrutura hierárquica. De forma análoga, os padrões e anti-padrões são classificados em categorias e sub-categorias. O modelo captura, ainda, os relacionamentos entre heurísticas, padrões e anti-padrões, descritos no capítulo 2.

Padrões e anti-padrões são os itens de conhecimento que podem ser detectados pela ferramenta. Esta detecção é realizada através de regras definidas em Prolog, onde cada padrão ou anti-padrão está associado a um predicado correspondente à cabeça de uma ou mais regras. Desta forma, os elementos *padrão* e *anti-padrão* são definidos no modelo como subclasses de *ItemDetectável* que, por sua vez, está relacionada com um elemento (*PredicadoProlog*) que define o predicado Prolog responsável pela sua detecção.

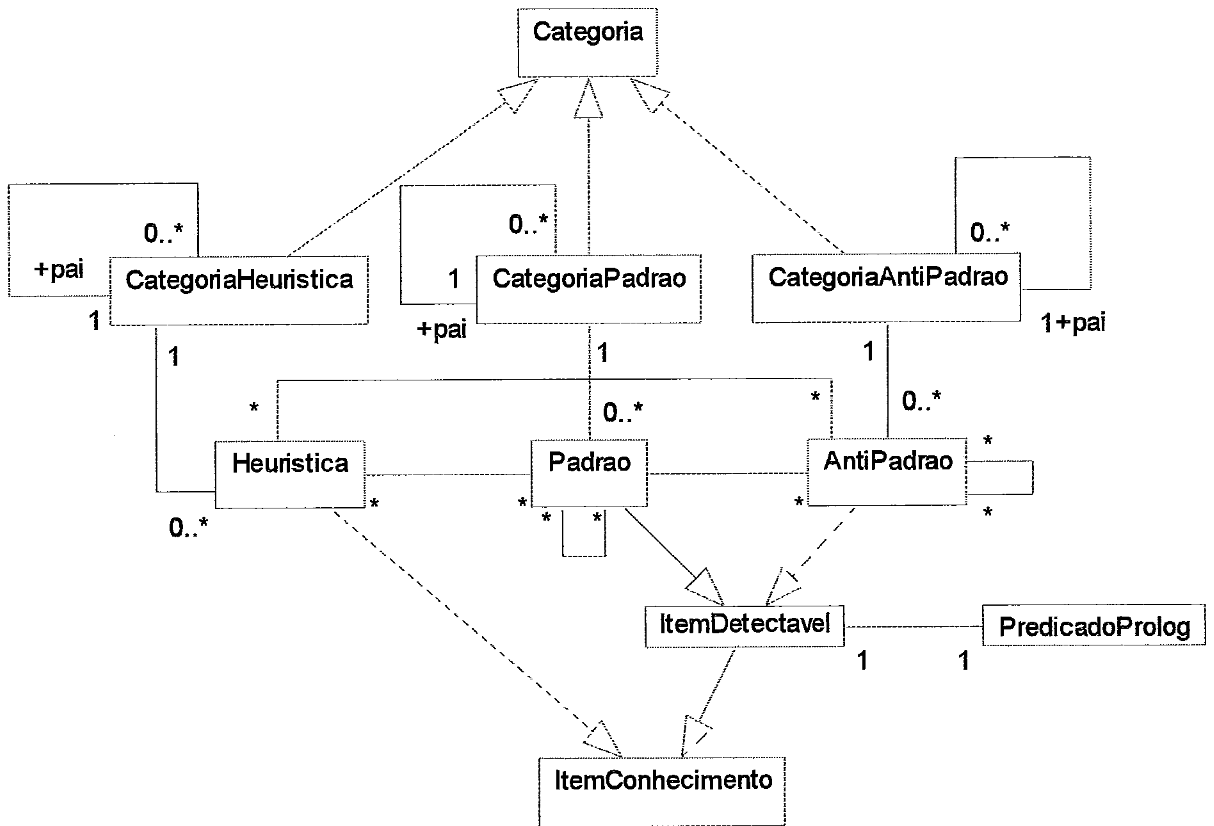


Figura 33 – OOPDTool - modelo de classes

A Figura 34 mostra a janela principal do OOPDTool. O menu *Conhecimento* disponibiliza o acesso à base de informações sobre heurísticas, padrões e anti-padrões através de três sub-menus. A opção *Atualiza Base* permite a geração da base de fatos Prolog que reflete as informações do projeto corrente no Rose. A opção *Detecção*, permite a detecção de anti-padrões e padrões neste projeto. Cada uma destas opções é descrita em detalhes nas próximas seções.

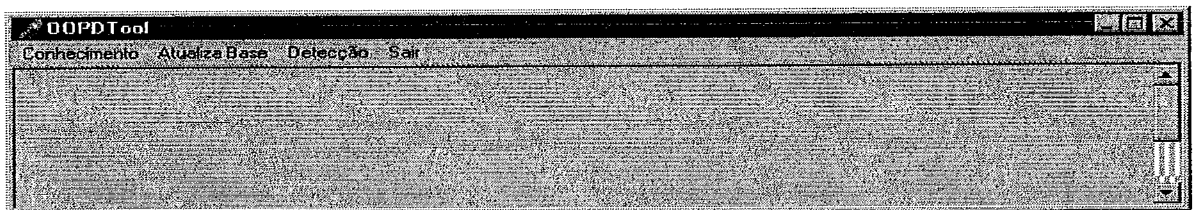


Figura 34: OOPDTool – menu principal

4.4. OOPDTool – Captura de Conhecimento

Este módulo corresponde à captura de heurísticas, padrões e anti-padrões de projeto orientado a objetos. Estes elementos podem ser catalogados a partir das publicações disponíveis na literatura e também da experiência dos próprios projetistas da organização.

4.4.1. Heurísticas

O acesso às heurísticas catalogadas é feito através da seleção do item de menu *"Conhecimento – Heurísticas"*. A ferramenta disponibiliza uma relação com as heurísticas catalogadas, conforme ilustrado na Figura 35. As heurísticas são agrupadas em categorias e sub-categorias definidas pelo usuário.

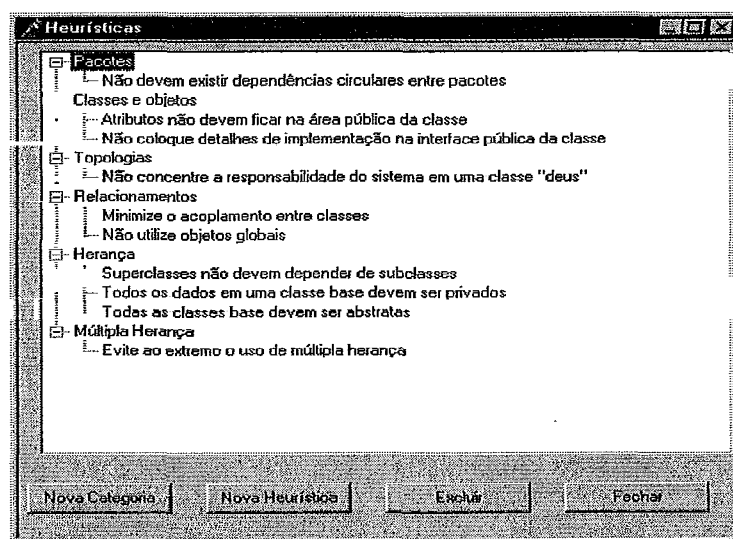


Figura 35: Catálogo de heurísticas

O botão *"Nova Categoria"* permite a criação de novas categorias e sub-categorias, na forma de uma árvore. Uma sub-categoria é inserida imediatamente abaixo de uma categoria ou sub-categoria selecionada. De modo análogo, o usuário pode criar uma nova heurística, bastando selecionar primeiro a categoria onde a heurística será inserida, pressionando, em seguida, o botão *"Nova Heurística"*, resultando na criação de uma heurística na categoria selecionada. Tanto as heurísticas como as categorias podem ser excluídas através do botão *"Excluir"*. Se o usuário selecionar uma categoria, todas as heurísticas e sub-categorias subordinadas a esta categoria são excluídas.

A edição de informações sobre uma heurística é realizada através de um duplo clique sobre a heurística desejada. A ferramenta disponibiliza um conjunto de janelas, conforme ilustrado na Figura 36, permitindo que o usuário detalhe as motivações, a fonte, e os relacionamentos desta heurística com possíveis violações (anti-padrões) e com soluções de projeto oriundas da aplicação desta heurística (padrões).

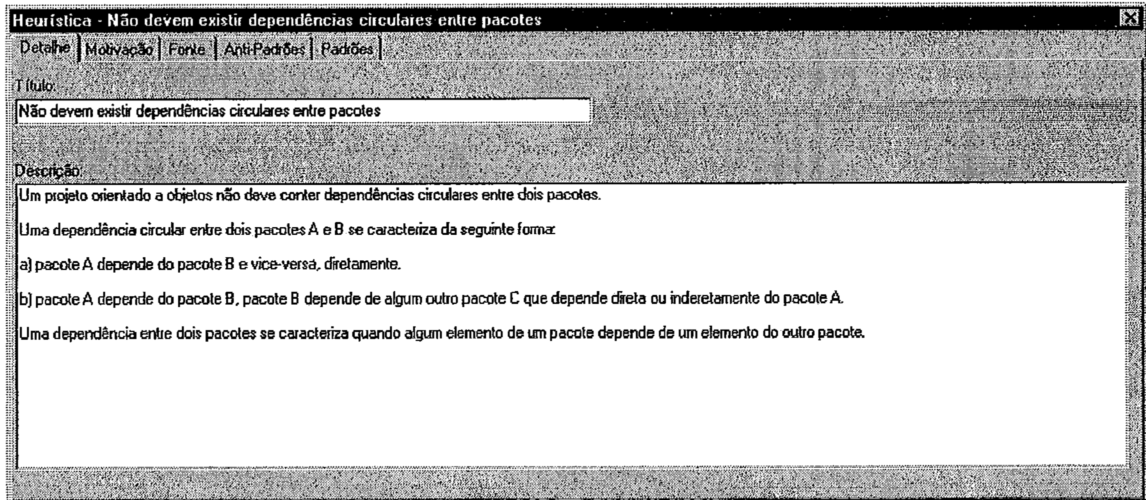


Figura 36: Janela de edição de uma heurística

A edição dos detalhes de uma heurística é realizada em cinco janelas tabuladas, correspondentes às seções para descrição de uma heurística, descritas no capítulo 3. Na primeira janela (Figura 36), o usuário define o nome da heurística e faz uma descrição sobre a mesma. Na janela "*Motivação*", são explicitados os motivos que justificam a existência desta heurística. Na janela "*Fonte*", o usuário pode indicar a origem desta heurística, ou seja, a referência bibliográfica ou uma pessoa da organização que tenha sido responsável pela sua definição.

Uma heurística pode estar associada a possíveis formas de violação, caracterizadas como anti-padrões. Na janela "*Anti-Padrões*" (Figura 37), o usuário pode fazer esta associação indicando um ou mais anti-padrões, ou seja, construções que caracterizem possíveis violações desta heurística. Para associar um anti-padrão à heurística em edição, basta o usuário selecioná-lo na lista à direita (Anti-padrões catalogados) e pressionar o botão "<<". Para remover a associação com um anti-padrão, basta selecioná-lo na lista à esquerda (Anti-padrões associados) e pressionar o botão ">>".

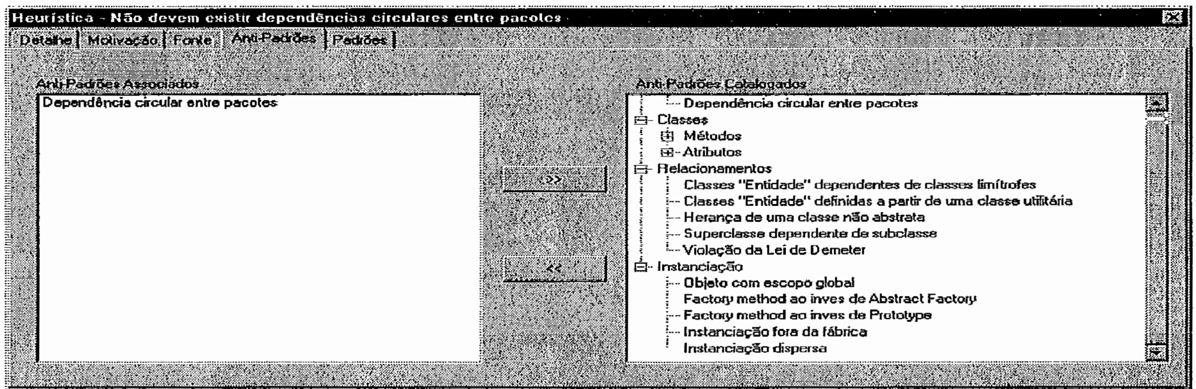


Figura 37: Relacionamento entre heurísticas e anti-padrões

O usuário pode definir associações entre as heurísticas e padrões de projeto, de forma análoga aos anti-padrões. A Figura 38 ilustra a janela onde esta associação pode ser realizada.

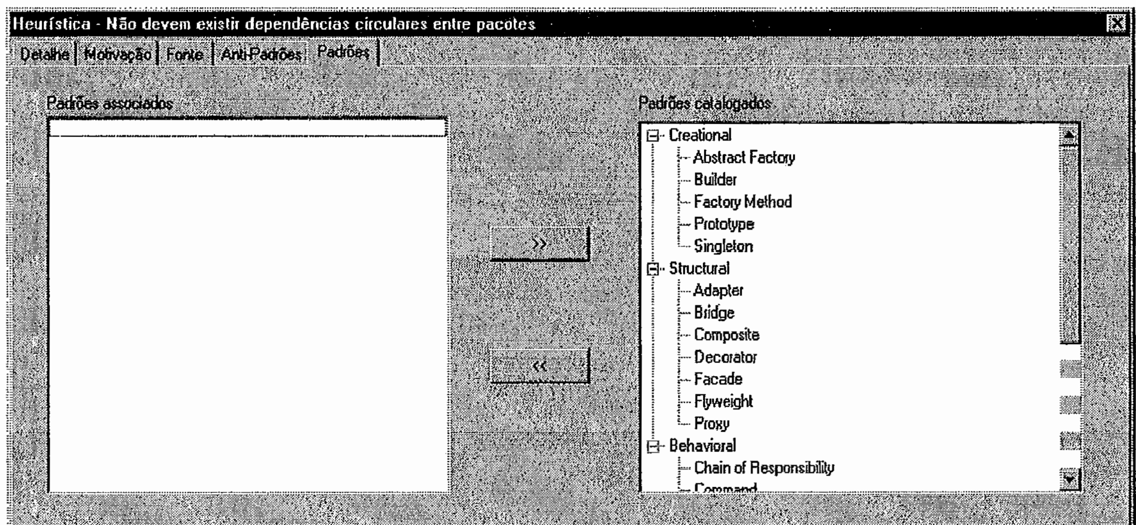


Figura 38: Relacionamento entre heurísticas e padrões de projeto

4.4.2. Padrões de Projeto

O acesso aos padrões catalogados é realizado através do item de menu *"Conhecimento – Padrões"*. A ferramenta disponibiliza uma relação com os padrões catalogados, conforme ilustrado na Figura 39. Os padrões são agrupados em categorias e sub-categorias definidas pelo usuário. O botão *"Nova Categoria"* permite a criação de novas categorias e sub-categorias de padrões. A criação de um novo padrão é feita selecionando-se primeiramente a categoria à qual o padrão pertence. Em seguida, o usuário pressiona o botão *"Novo Padrão"*. O novo padrão é incluído na categoria selecionada previamente. Para excluir uma categoria e os respectivos padrões, basta o usuário selecionar uma categoria e pressionar o botão *"Excluir"*. Para excluir apenas um padrão, basta selecionar apenas o padrão desejado antes de pressionar o botão *"Excluir"*.

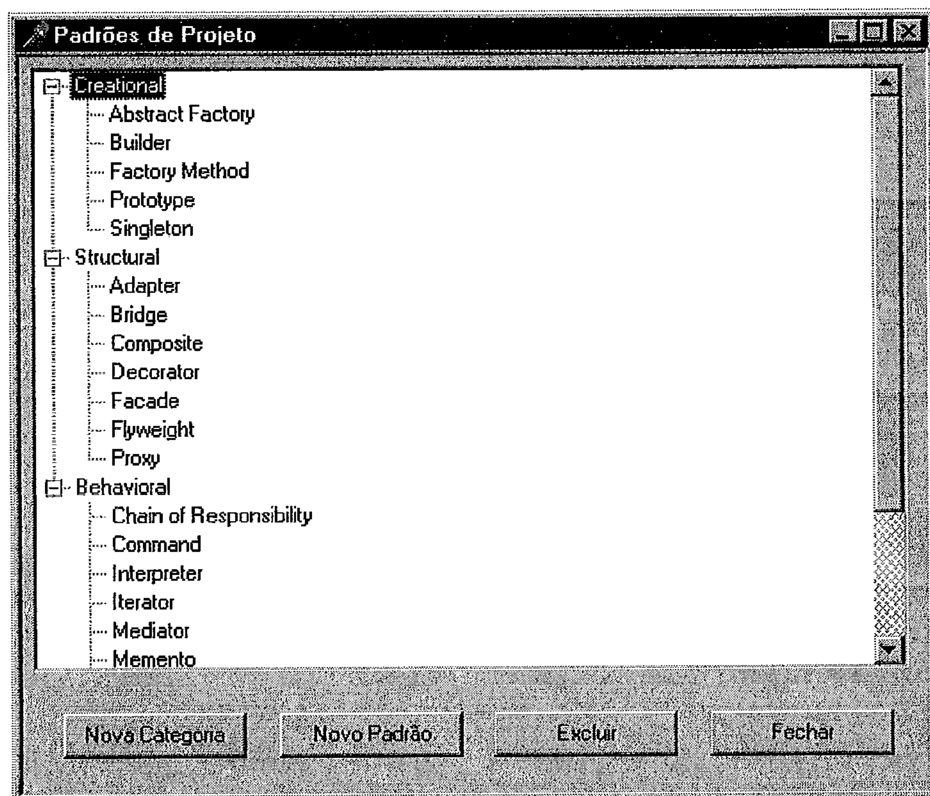


Figura 39: Catálogo de padrões

A edição de informações sobre um padrão é realizada através de um duplo clique no padrão desejado. A ferramenta disponibiliza um conjunto de janelas, conforme ilustrado na Figura 40, permitindo que o usuário detalhe as informações sobre o padrão,

baseando-se nas seções que o compõem, descritas no capítulo 3.

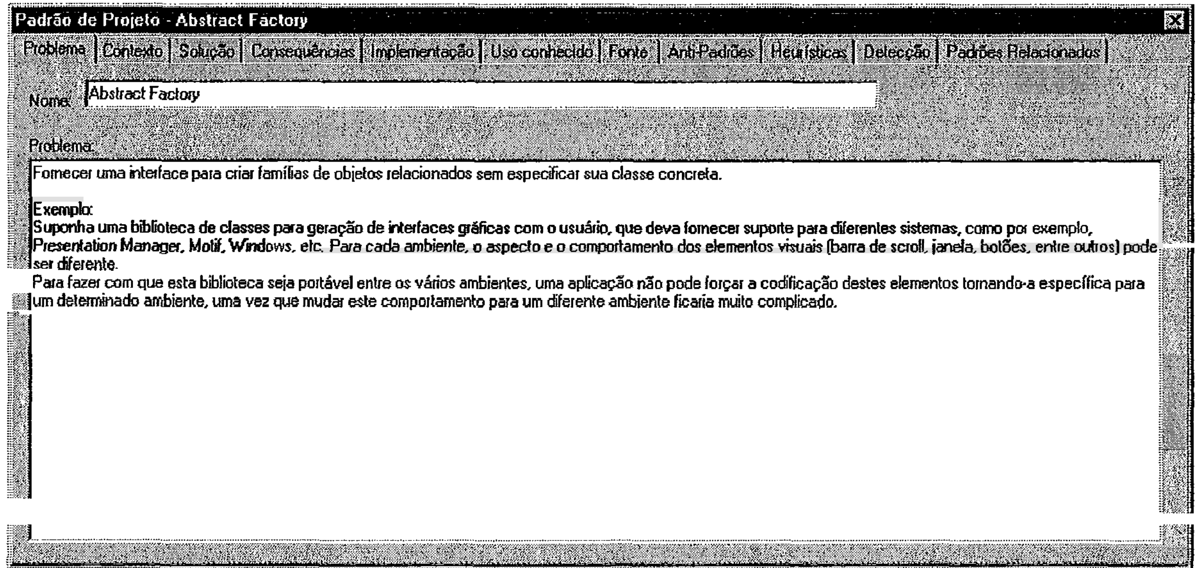


Figura 40: Janela de edição de padrão

As primeiras janelas correspondem às seções que definem um padrão. Nestas janelas, o usuário define o nome do padrão, o problema que este se propõe a resolver, o contexto de sua aplicação, a solução proposta, as conseqüências de sua aplicação, detalhes sobre as diferentes alternativas de implementação da solução, onde este padrão já foi utilizado, e qual a sua fonte, ou seja, os responsáveis pela sua criação.

Na janela ilustrada na Figura 41, o usuário pode associar anti-padrões ao padrão em questão. Desta forma, é possível catalogar soluções ruins de projeto que poderiam ser substituídas pela aplicação de um padrão. Normalmente, estas soluções ruins são empregadas em função do projetista desconhecer a existência de um determinado padrão. O campo *"Comentário"* permite que sejam adicionados detalhes sobre a associação entre um padrão e anti-padrões.

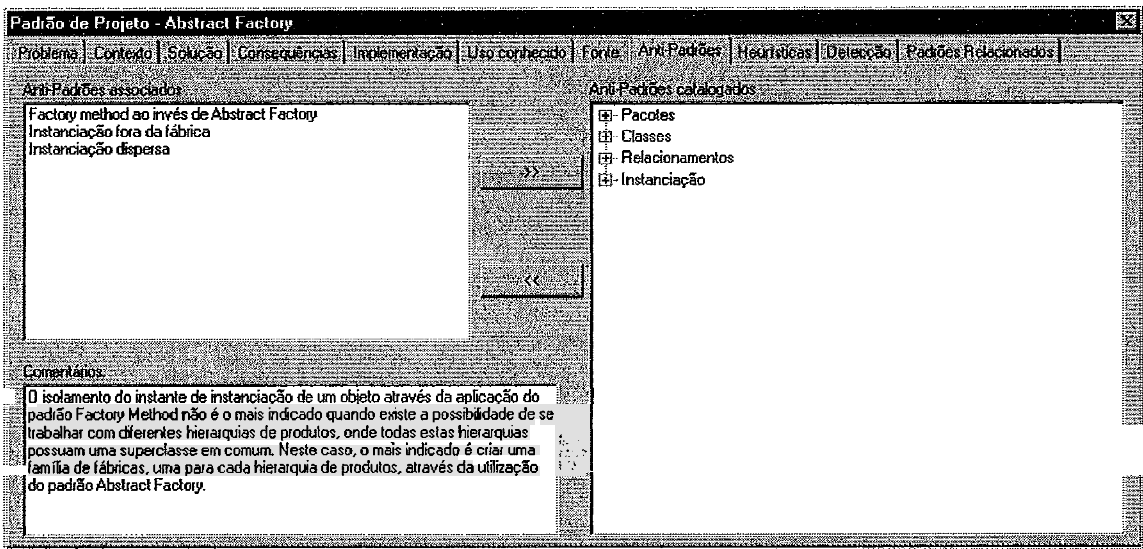


Figura 41 : Associação de anti-padrões a um padrão

De forma análoga, o usuário pode associar heurísticas ao padrão em edição. Desta forma, o projetista pode tornar explícito que heurísticas estão por trás do emprego de um determinado padrão.

Na janela *"Detecção"* (Figura 42), são definidos alguns parâmetros para a detecção do padrão. Um padrão é detectado através de uma ou mais regras escritas em Prolog, onde cada regra captura uma possível forma de detecção do padrão.

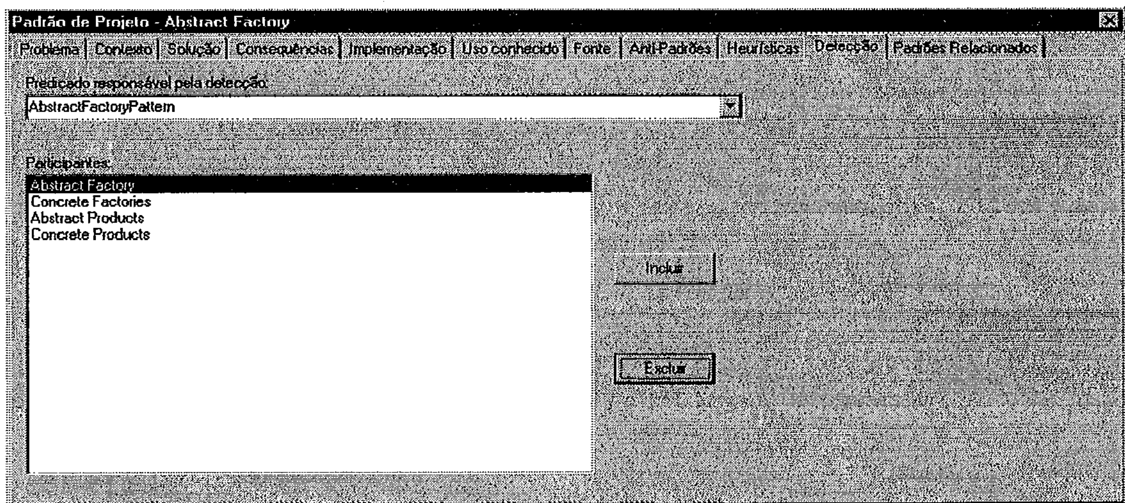


Figura 42: Parâmetros para a detecção de um padrão

Uma regra é definida pela forma geral *"cabeça :- corpo"*, ou seja, se as condições estabelecidas em *"corpo"* forem satisfeitas, então podemos concluir que *"cabeça"* é verdadeiro. Todas as regras que definem um padrão devem possuir o mesmo nome na

cabeça, ou seja, o conjunto de todas as possíveis formas de detecção de um padrão devem sempre gerar, como conclusão, o mesmo predicado *"cabeça"*. As regras para a detecção dos padrões são definidas, armazenadas e compiladas em uma base de regras, utilizando o Visual Prolog 5.

A Figura 43 ilustra uma regra para captura do padrão *"AbstractFactory"*. Todas as possíveis regras para captura deste padrão devem ser definidas tendo como *"cabeça"* da regra o predicado *abstractFactoryPattern (AbstractFactory, ConcreteFactories, AbstractProducts, ConcreteProducts).*

```
abstractFactoryPattern(AbstractFactory,ConcreteFactories,AbstractProducts,ConcreteProducts):-
        abstractFactory(AbstractFactory,AbstractProducts),
        findAll(ConcreteFactory,concreteFactory(ConcreteFactory,AbstractFactory), ConcreteFactories),
         listSize(ConcreteFactories, NumFact), NumFact > 0,
        findAll (X, concreteProduct (X, AbstractProducts), ConcreteProducts).
```

Figura 43: Regra para detecção do padrão "*AbstractFactory*"

Nesta janela, é feita uma associação de um padrão com o predicado *cabeça* que define as possíveis formas de detectá-lo. Além de informar o predicado a ser utilizado no processo de detecção, o usuário pode, ainda, definir como as informações sobre os componentes de um padrão serão exibidos para o usuário. Isto é feito através da caixa *"Participantes"*, onde cada participante corresponde a uma informação retornada pela regra de detecção. No mesmo exemplo da Figura 43, os participantes do padrão correspondem às informações *"AbstractFactory"*, *"ConcreteFactories"*, *"AbstractProducts"* e *"ConcreteProducts"* que serão retornadas pelo predicado *"abstractFactoryPattern"* no caso de uma eventual detecção do padrão no projeto. Os nomes dos participantes colocados nesta janela correspondem a um cabeçalho informativo de cada uma destas informações. Este cabeçalho será utilizado na exteriorização do resultado do processo de detecção de padrões, de forma a permitir que o projetista saiba quais elementos do modelo de projeto se encaixam em cada um dos papéis definidos no padrão.

Finalizando a definição de um padrão, é possível catalogar também possíveis padrões relacionados ao padrão em edição, como ilustrado na Figura 44. O padrão *"AbstractFactory"*, por exemplo, está relacionado com os padrões *"Singleton"*,

"FactoryMethod" e *"Prototype"*, uma vez que sua implementação é normalmente feita através da aplicação destes padrões relacionados. O campo *"Comentário"* permite que sejam adicionados detalhes sobre esta associação.

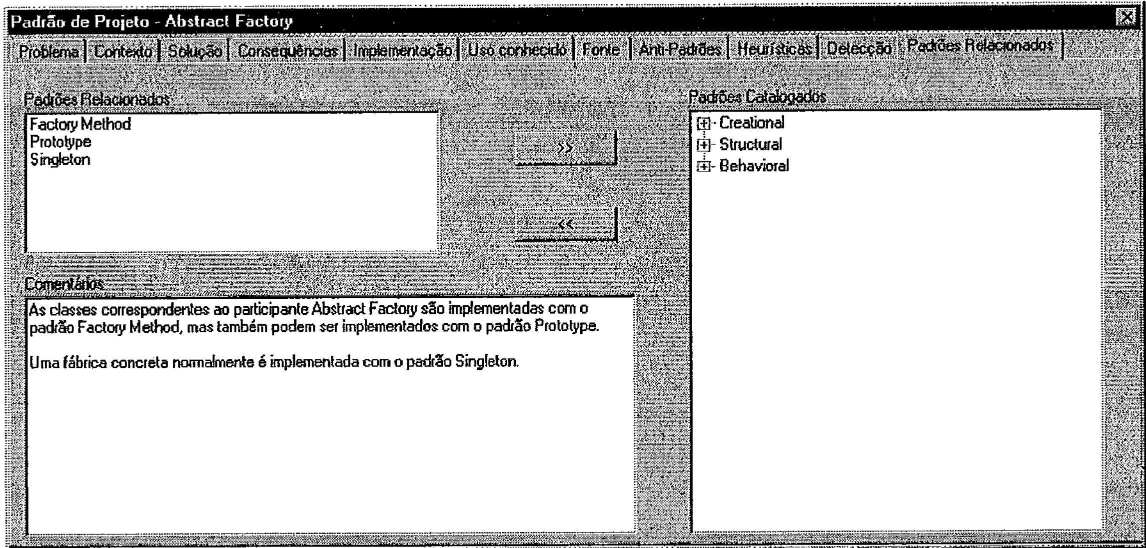


Figura 44: Associação entre padrões

4.4.3. Anti-Padrões

O acesso aos anti-padrões catalogados é realizado através do item de menu *"Conhecimento – Anti-Padrões"*. A ferramenta disponibiliza uma relação com os anti-padrões catalogados, conforme ilustrado na figura Figura 45. A manutenção desta relação é realizada de forma análoga às heurísticas e padrões.

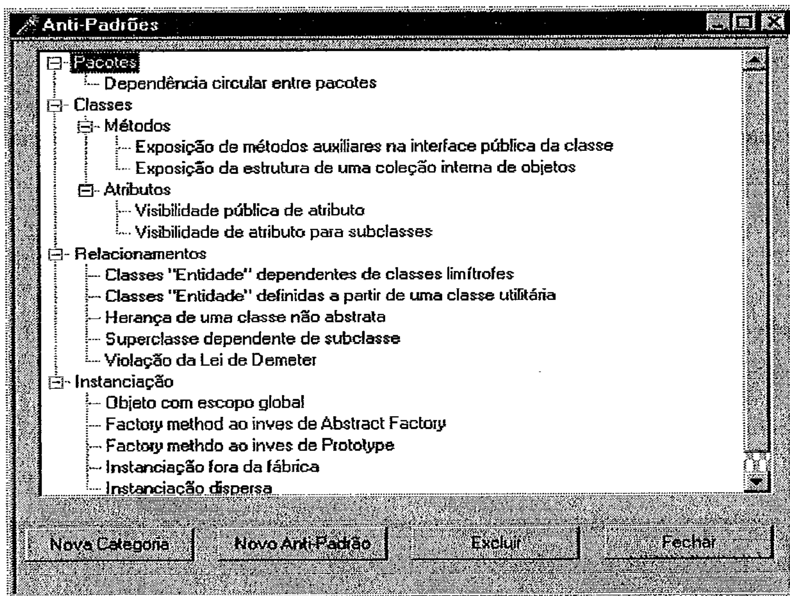


Figura 45: Catálogo de anti-padrões

A edição de informações sobre um anti-padrão é realizada através de um duplo clique sobre o anti-padrão desejado. A ferramenta disponibiliza um conjunto de janelas, conforme ilustrado na Figura 46, permitindo que o usuário detalhe estas informações baseando-se nas seções que compõem um anti-padrão, descritas no capítulo 3.

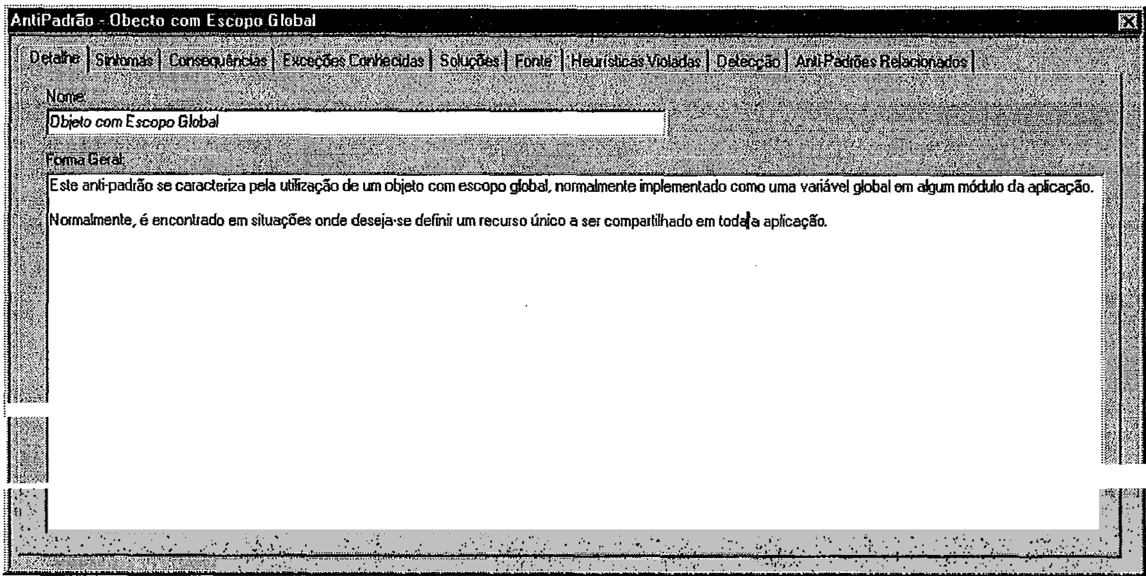


Figura 46: Janela de edição de anti-padrões

As primeiras janelas correspondem às seções que definem um anti-padrão. Nestas janelas, o usuário define o nome do anti-padrão, a forma geral de ocorrência deste anti-padrão, os sintomas e as conseqüências de sua utilização, possíveis exceções, uma vez que uma construção normalmente reconhecida como uma solução *ruim* pode eventualmente ser aplicada em situações excepcionais. Uma característica importante de um anti-padrão é como transformá-lo em uma boa construção. As possíveis formas de realizar esta transformação são detalhadas na janela "*Soluções*", conforme ilustrado na Figura 47. Muitas vezes, esta transformação consiste na aplicação de um padrão conhecido, como por exemplo, a substituição de objetos globais pela aplicação do padrão "*Singleton*". No combo-box da parte inferior desta janela, é possível informar ao projetista que a solução para este anti-padrão deve ser feita através da aplicação de um padrão de projeto. O botão "*Detalhe*" permite que o usuário possa visualizar a definição deste padrão associado.

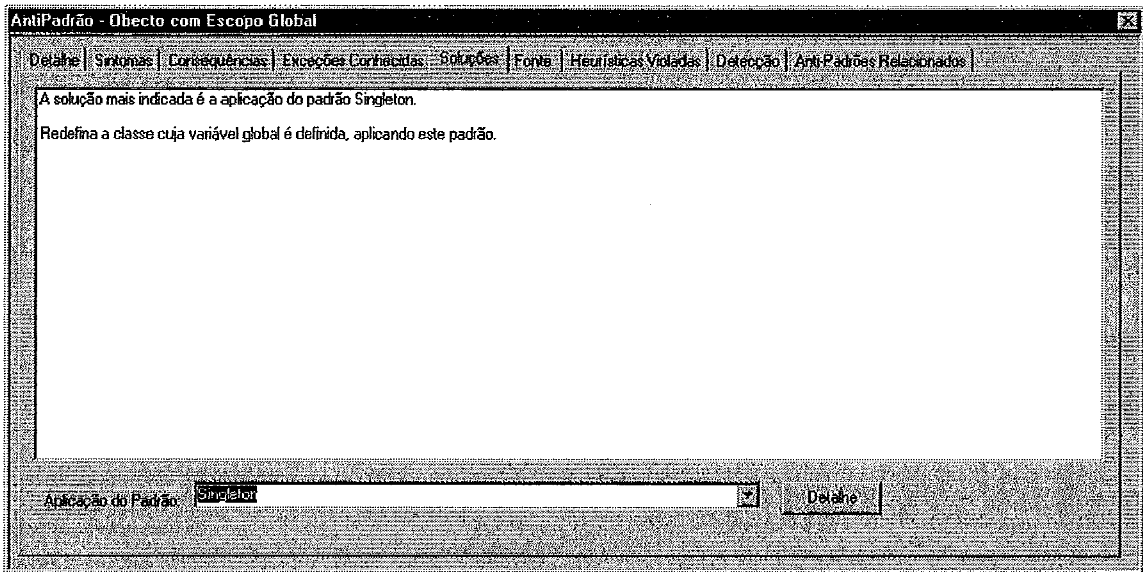


Figura 47: Anti-padrões - Soluções

A janela "Fonte" indica a origem deste anti-padrão, seja através de uma referência bibliográfica ou ainda através da referência a uma pessoa ou setor da organização responsável pela definição do anti-padrão.

Um anti-padrão pode ainda estar relacionado com heurísticas, no sentido de corresponder a possíveis violações das mesmas. A Figura 48 ilustra a janela onde é possível visualizar e estabelecer as heurísticas associadas a este anti-padrão. Vale ressaltar que associações entre heurísticas e anti-padrões podem ser feitas tanto nesta janela como na janela de anti-padrões associados a uma heurística (Figura 37).

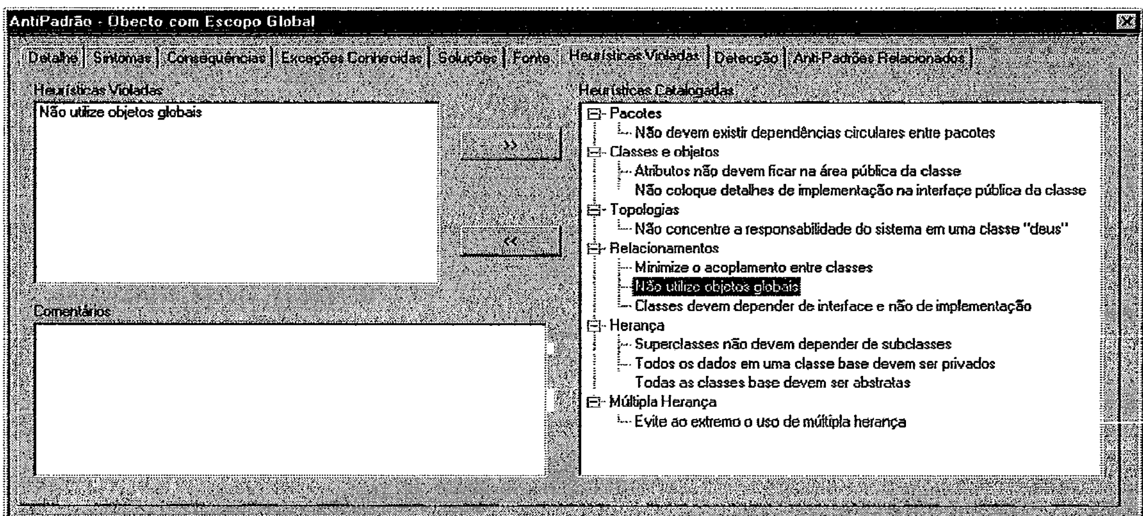


Figura 48: Heurísticas associadas a um anti-padrão

Um anti-padrão é detectado de forma análoga a um padrão, ou seja, através de uma ou mais regras escritas em Prolog, onde cada regra captura uma possível forma de detecção do anti-padrão. Desta forma, possíveis variações na estrutura ou dinâmica dos elementos que compõem o anti-padrão podem ser capturadas. As regras utilizadas para a detecção dos anti-padrões são definidas, armazenadas e compiladas em uma base de regras em Prolog para detecção de anti-padrões.

Na janela Detecção (Figura 49) são definidos alguns parâmetros para a detecção do anti-padrão. Nesta janela, é feita uma associação de um anti-padrão com o predicado *cabeça* que define as possíveis formas de detectá-lo. Além de informar o predicado a ser utilizado no processo de detecção, o usuário pode, ainda, definir como as informações sobre os componentes de um anti-padrão serão exibidos para o usuário. Isto é feito através da caixa "*Participantes*", onde cada participante corresponde a uma informação retornada pela regra de detecção, de forma análoga à detecção dos padrões.

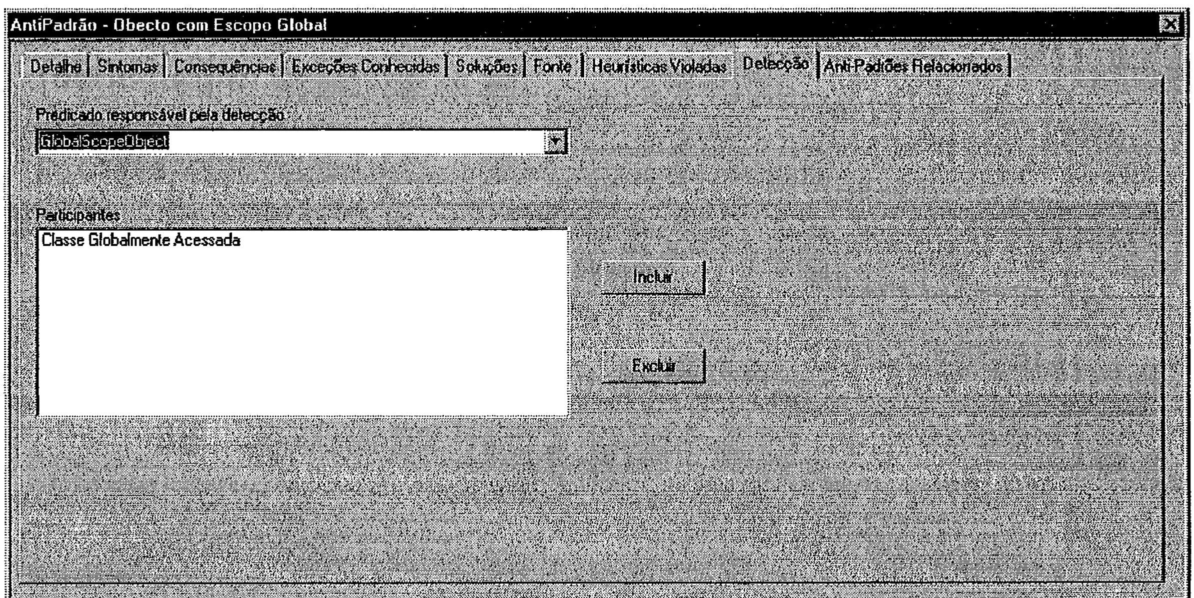


Figura 49: Detecção de um anti-padrão

Finalizando a definição de um anti-padrão, é possível catalogar também anti-padrões relacionados ao padrão em edição. A solução isolada de um anti-padrão pode, eventualmente, levar a um outro anti-padrão. Suponha, por exemplo, o anti-padrão *"Visibilidade Pública de Atributo"* correspondente à definição de atributos na área pública de uma classe. Uma construção mais adequada seria a definição de métodos de

acesso que encapsulassem o acesso a estes atributos. Entretanto, se a configuração final desta classe contiver apenas estes métodos de acesso, além de métodos construtores e destrutores, isto pode ser sinal de que o anti-padrão *"Blob"*, descrito no capítulo 2, esteja presente.

4.5. OOPDTool – Gerador de Fatos

Uma vez que as heurísticas, padrões e anti-padrões estejam devidamente catalogados, utilizando o módulo *Captura de Conhecimento* descrito anteriormente, e que as regras para a detecção dos padrões e anti-padrões tenham sido definidas, registradas e compiladas em Prolog, o projetista passa a ter à sua disposição a possibilidade de detectar estas construções em um projeto orientado a objetos elaborado no Rose.

Esta detecção é realizada a partir das informações do projeto corrente na ferramenta CASE. Como o Rose, originalmente, não fornece suporte para a captura de todas as informações requeridas pelo OOPDTool, conforme o metamodelo descrito no capítulo 3, foi utilizado um outro recurso de extensão do Rose que permite definir propriedades adicionais para cada um dos principais elementos do modelo, ou seja, pacotes, classificadores, atributos e operações. Cada ferramenta adicionada ao Rose pode definir seu próprio conjunto de propriedades para cada um dos elementos do modelo. Desta forma, ao adicionarmos a OOPDTool como uma extensão do Rose, definimos, para cada elemento, um conjunto de propriedades correspondendo a algumas informações presentes no metamodelo mas não capturadas pelo Rose. Cada propriedade tem um nome, podendo corresponder a um valor booleano, numérico, string ou ainda, um valor de uma enumeração.

A Figura 50 ilustra uma extensão realizada para capturar informações adicionais sobre uma operação de um classificador. A propriedade *"Escopo"* corresponde ao escopo da operação (instância ou classe), *"Polimorfica"* corresponde à possibilidade de redefinição da operação nas subclasses, *"Abstrata"* indica se a operação possui ou não uma implementação e *"Const"* define se a operação mantém o estado do objeto após a sua invocação. De forma análoga, foram definidas algumas propriedades adicionais para

atributos e classificadores que originalmente não são fornecidas pelo Rose.

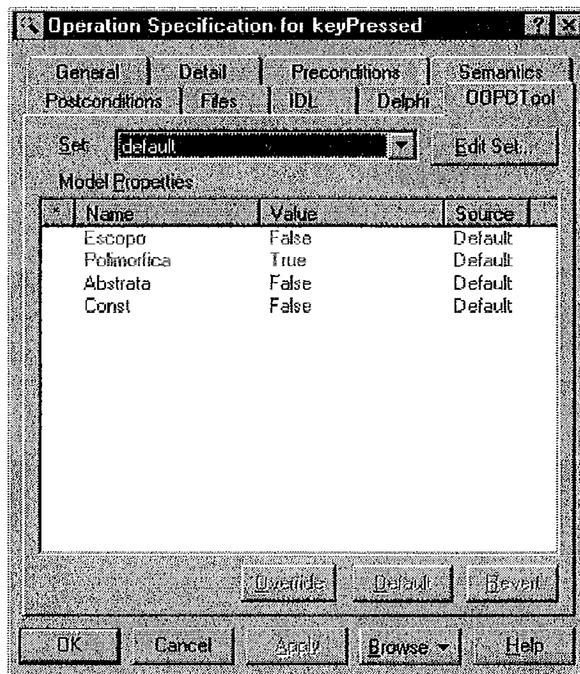


Figura 50: Extensão do Rose pela adição de propriedades

Uma característica importante para a detecção de padrões que envolvam a parte de dinâmica de colaboração entre objetos é o modo com que estas informações devem ser montadas no Rose para que o OOPDTool possa funcionar corretamente. Toda operação de uma classe que invoque operações de objetos (da mesma classe ou de outras) deve formalizar esta necessidade de interação através de um diagrama de colaboração. Este diagrama contempla todas as invocações diretamente realizadas pela operação, a classe de cada objeto envolvido nestas chamadas, bem como a visibilidade de cada um destes objetos (global, local, parâmetro, atributo ou local indireto). Segundo BOOCH et al. (1999), a elaboração de diagramas de colaboração para o contexto de cada operação abre ainda a possibilidade para a geração de código a partir do próprio diagrama, hoje limitada, na maioria das ferramentas comercialmente disponíveis, apenas à geração de esqueleto de código a partir das definições estáticas das classes.

A Figura 51 ilustra um exemplo onde é feita a associação de um diagrama de colaboração a uma operação de uma classe.

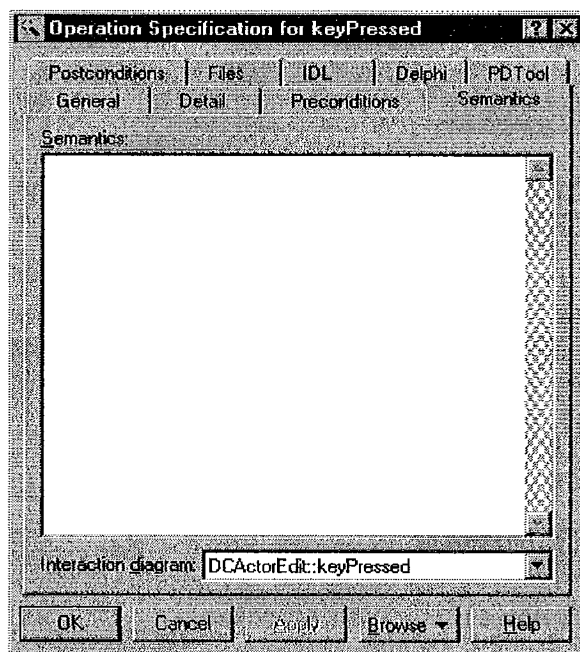


Figura 51 : Associação de um diagrama de colaboração a um método de uma classe

A Figura 52 ilustra um exemplo de diagrama correspondente à colaboração necessária para a execução de um método de uma classe. Neste exemplo é apresentado o diagrama de colaboração para o método *keyPressed* da classe *DCActorEdit* do editor de diagramas (WERNER et al., 1999). Neste diagrama são especificadas apenas as chamadas diretamente realizadas por este método. A primeira chamada corresponde à invocação da operação *getKeyCode* a um objeto da classe *KeyEvent* recebido como parâmetro. A terceira mensagem corresponde à invocação da operação *DeleteSelectedObjects* do objeto *diagramBox* da classe *DCDiagramBox*. Este objeto é um atributo da superclasse *DCActor*, obtido através da operação *GetDiagramBox*. Finalmente, a última mensagem corresponde à invocação da operação *Repaint* do objeto *diagramBox*. O desencadeamento de uma determinada chamada em outras invocações de operações são especificadas nos diagramas correspondentes às mesmas, ou seja, se uma operação A invocar uma operação B que, por sua vez, invocar uma operação C, o diagrama de colaboração correspondente à operação A conterá apenas a invocação da operação B, enquanto que a invocação da operação C estará presente no diagrama de

colaboração correspondente à operação B.

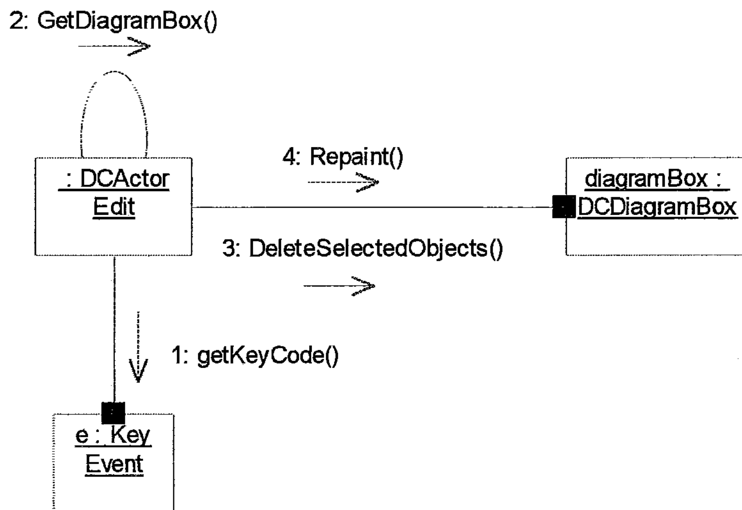


Figura 52: Diagrama de colaboração para uma operação

A ferramenta OOPDTool realiza o processo de detecção de padrões e anti-padrões a partir de uma base de fatos em Prolog, correspondente aos elementos que descrevem as construções existentes no modelo elaborado no Rose. Para que esta base não precise ser gerada a cada pedido de detecção de padrões ou anti-padrões, existe uma opção no menu principal "*Atualizar Base*", através da qual o usuário pode sincronizar o conteúdo desta base com o projeto aberto no Rose. Esta sincronização corresponde a um módulo do OOPDTool escrito em Delphi que acessa as informações do modelo do Rose via automação OLE, e atualiza a base de fatos que será objeto de análise pelo módulo de detecção de padrões e anti-padrões. A Figura 54 mostra um pequeno extrato da base de fatos gerada através desta opção, correspondente aos modelos das Figura 52 e Figura 53.

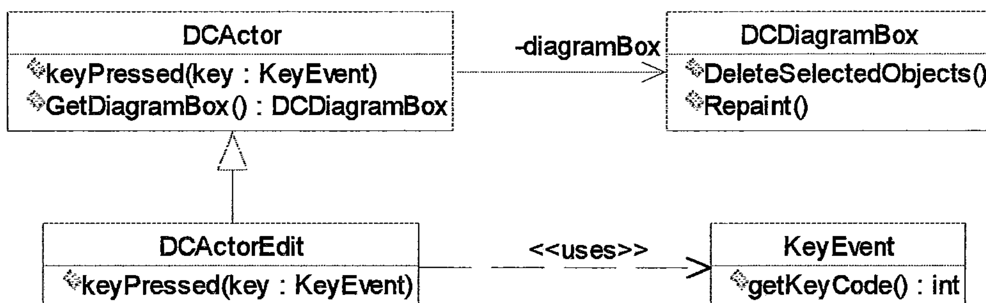


Figura 53: Extrato do diagrama de classes do editor de diagramas

```
pacote ("LogicalView", "Nil", "Nil").

classificador ("LogicalView", "LogicalView::DCActor", "Nil", "Classe", "Abstrata", "NaoFolha", "NaoRaiz").
classificador ("LogicalView", "LogicalView::DCDiagramBox", "Nil", "Classe", "Concreta", "NaoFolha", "NaoRaiz").
classificador ("LogicalView", "LogicalView::KeyEvent", "Nil", "Classe", "Concreta", "NaoFolha", "NaoRaiz").
classificador ("LogicalView", "LogicalView::DCActorEdit", "Nil", "Classe", "Concreta", "NaoFolha", "NaoRaiz").

visibilidadeNoPacote ("LogicalView", "LogicalView::DCActor", "Public").
visibilidadeNoPacote ("LogicalView", "LogicalView::DCDiagramBox", "Public").
visibilidadeNoPacote ("LogicalView", "LogicalView::KeyEvent", "Public").
visibilidadeNoPacote ("LogicalView", "LogicalView::DCActorEdit", "Public").

herdaDe ("LogicalView::DCActorEdit", "LogicalView::DCActor").

atributo ("LogicalView::DCActor", "LogicalView::DCActor::diagramBox", "Instancia", "Private",
"LogicalView::DCDiagramBox", "LogicalView::DCDiagramBox", "NonConst", "1", "Association").

operacao ("LogicalView::DCActor", "LogicalView::DCActor::keyPressed", "keyPressed", "Instancia", "Public", "Nil",
"Virtual", "Abstrata", "NonConst").
operacao ("LogicalView::DCActor", "LogicalView::DCActor::GetDiagramBox", "GetDiagramBox", "Instancia", "Public",
"Nil", "Virtual", "Metodo", "NonConst").
operacao ("LogicalView::DCDiagramBox", "LogicalView::DCDiagramBox::DeleteSelectedObjects",
"DeleteSelectedObjects", "Instancia", "Public", "Nil", "Virtual", "Metodo", "NonConst").
operacao ("LogicalView::DCDiagramBox", "LogicalView::DCDiagramBox::Repaint", "Repaint", "Instancia", "Public", "Nil",
"Virtual", "Metodo", "NonConst").
operacao ("LogicalView::KeyEvent", "LogicalView::KeyEvent::getKeyCode", "getKeyCode", "Instancia", "Public", "Nil",
"Virtual", "Metodo", "NonConst").
operacao ("LogicalView::DCActorEdit", "LogicalView::DCActorEdit::keyPressed", "keyPressed", "Instancia", "Public",
"Nil", "Virtual", "Metodo", "NonConst").

parametro ("LogicalView::DCActor::keyPressed", "LogicalView::DCActor::keyPressed::key", " 1", "In",
"LogicalView::KeyEvent").
parametro ("LogicalView::DCActor::GetDiagramBox", "LogicalView::DCActor::GetDiagramBox::Return", " 1", "Return",
"LogicalView::DCDiagramBox").
parametro ("LogicalView::KeyEvent::getKeyCode", "LogicalView::KeyEvent::getKeyCode::Return", " 1", "Return", "int").
parametro ("LogicalView::DCActorEdit::keyPressed", "LogicalView::DCActorEdit::keyPressed::key", " 1", "In",
"LogicalView::KeyEvent").

invoca ("LogicalView::DCActorEdit::keyPressed", "LogicalView::KeyEvent", "LogicalView::KeyEvent::getKeyCode",
"Parametro").
invoca ("LogicalView::DCActorEdit::keyPressed", "LogicalView::DCActorEdit", "LogicalView::DCActor::GetDiagramBox",
"Self").
invoca ("LogicalView::DCActorEdit::keyPressed", "LogicalView::DCDiagramBox",
"LogicalView::DCDiagramBox::DeleteSelectedObjects", "Atributo").
invoca ("LogicalView::DCActorEdit::keyPressed", "LogicalView::DCDiagramBox",
"LogicalView::DCDiagramBox::Repaint", "Atributo").
```

Figura 54: Extrato da base de fatos gerada pelo OOPDTool

4.6. OOPDTool - Analisador

Este módulo disponibiliza facilidades para a detecção de padrões e anti-padrões em um projeto orientado a objetos. Os padrões e anti-padrões devem estar previamente catalogados no OOPDTool, assim como as respectivas regras para sua detecção. O processo de detecção é realizado verificando-se quais regras, correspondentes a padrões ou anti-padrões, são satisfeitas pelos fatos que descrevem os elementos presentes no projeto que está sendo objeto de análise.

A detecção de anti-padrões pode ser realizada a partir de três modos:

- através de seleção direta na relação de anti-padrões catalogados: a partir do menu principal, a opção de menu *"Detecção – Anti-Padrões – Diretamente"* dá acesso a este modo de seleção, ilustrado na Figura 55. O usuário pode selecionar os anti-padrões diretamente na lista, ou selecionar todos a partir do botão "*Selecionar Tudo*". Uma seleção pode ser cancelada através do botão "*Cancelar Tudo*". Esta janela disponibiliza, ainda, uma opção para a visualização da definição de um anti-padrão. Para tanto, basta o usuário selecionar o anti-padrão desejado e pressionar o botão "*Definição*". Para iniciar a busca pelos anti-padrões selecionados, basta selecionar o botão "*Detectar*". A detecção dos anti-padrões pode abranger todo o modelo ou apenas os elementos selecionados em um diagrama do projeto corrente no Rose, bastando indicar a opção desejada, através dos botões "*Modelo completo*" e "*Somente os elementos selecionados*" (Figura 55). Ao final do processo de detecção, surge, ao lado de cada anti-padrão detectado, um número entre colchetes indicando o número de ocorrências deste anti-padrão no escopo da busca, conforme ilustrado na Figura 55.

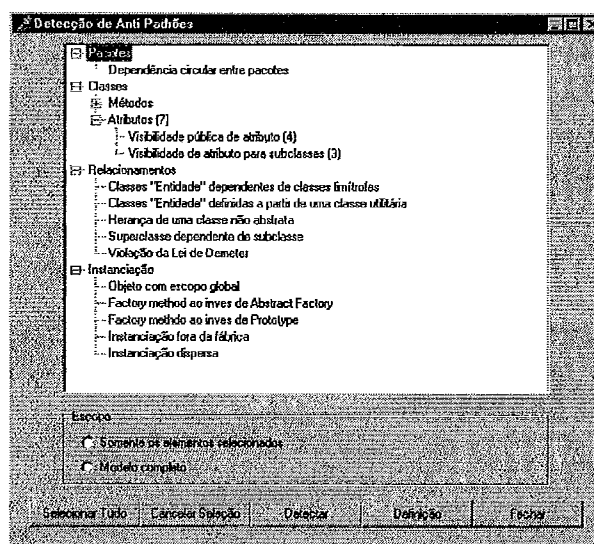


Figura 55: Seleção direta de anti-padrões

- através da seleção a partir das heurísticas catalogadas: a partir do menu principal, a opção de menu *"Detecção – Anti-Padrões – Via Heurísticas"* dá acesso a este modo de seleção, ilustrado na Figura 56. Através deste modo de

seleção, é possível detectar anti-padrões relacionados a um determinado conjunto de heurísticas, ou seja, o foco está em detectar possíveis violações destas heurísticas através dos anti-padrões associados. A seleção e o cancelamento da seleção é feito de modo análogo ao descrito anteriormente. Para iniciar a busca pelos anti-padrões associados às heurísticas selecionadas, basta selecionar o botão *"Detectar"*. Ao final do processo de detecção, surge, ao lado de cada heurística violada, um número entre colchetes indicando o número de violações detectadas.

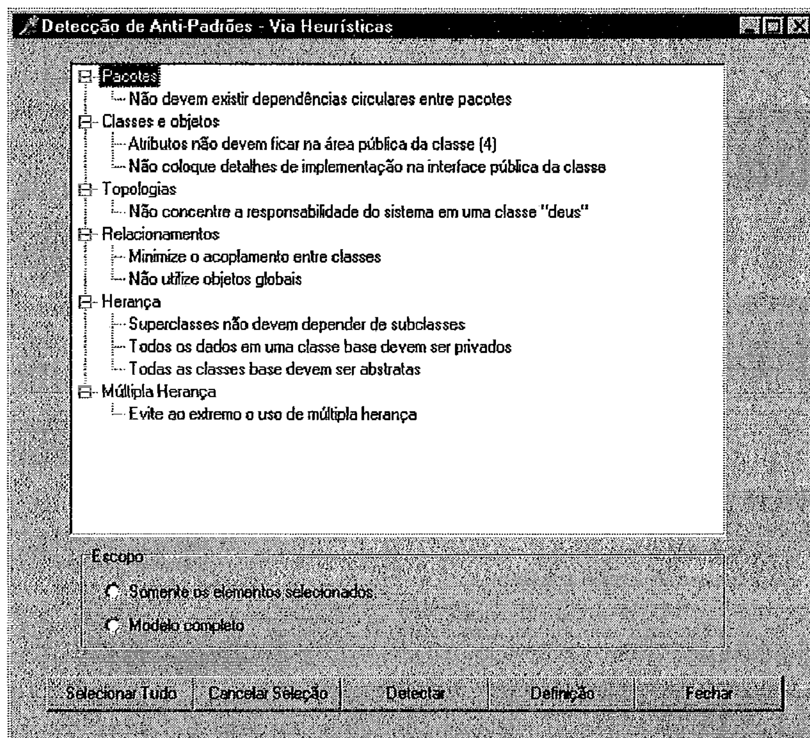


Figura 56: Seleção via heurísticas

- através da seleção a partir dos padrões catalogados: a partir do menu principal, a opção de menu *"Detecção – Anti-Padrões – Via Padrões"* dá acesso a este modo de seleção, ilustrado na Figura 57. Esta forma de seleção permite que a detecção seja dirigida com o objetivo de encontrar situações onde um padrão de projeto poderia ter sido aplicado. A forma de seleção é análoga ao da seleção através de heurísticas. Ao final do processo de detecção, aparece ao lado de cada padrão um número entre colchetes indicando o número de anti-padrões detectados associados a cada padrão.

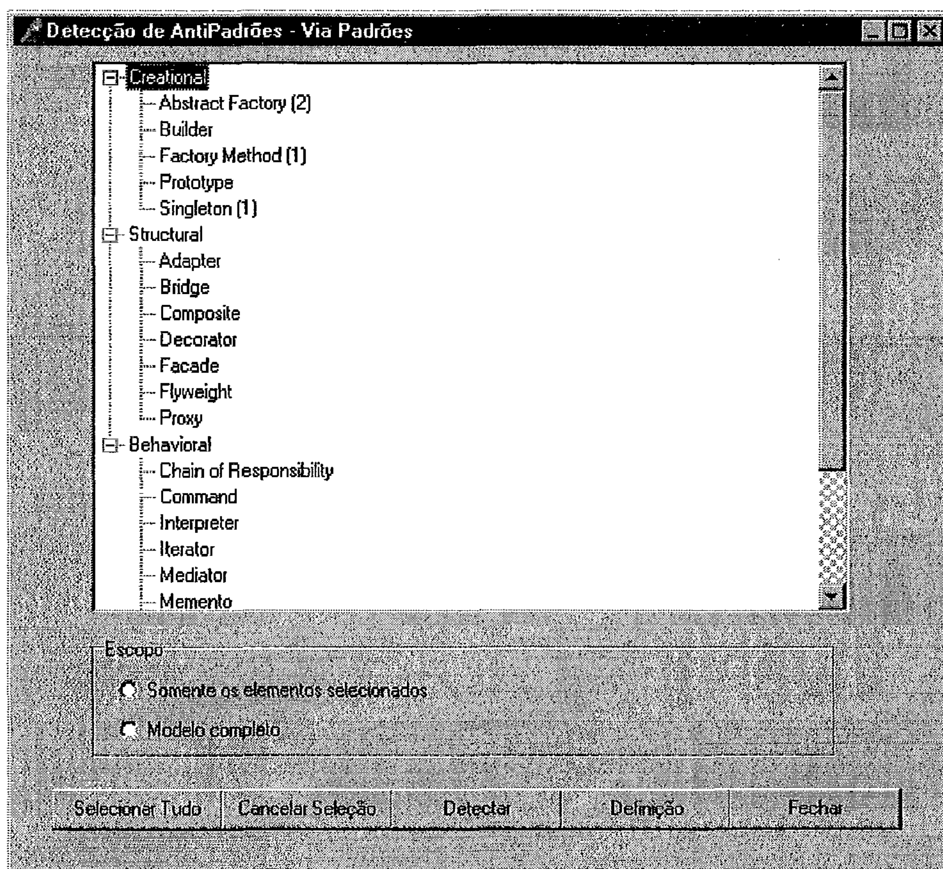


Figura 57: Seleção via padrões

A partir da lista de anti-padrões detectados, independente do modo de seleção (diretamente pelo catálogo de anti-padrões, pelo catálogo de heurísticas, ou ainda pelo catálogo de padrões), o usuário tem acesso aos detalhes das ocorrências de cada anti-padrão. Para isto, basta dar um duplo clique em algum anti-padrão, cujo número de ocorrências esteja indicado entre colchetes. A ferramenta disponibiliza, então, um relatório indicando todas as ocorrências dos anti-padrões no modelo ou fragmento de modelo selecionado no Rose. A Figura 58 ilustra um exemplo resultante da detecção de algumas ocorrências do anti-padrão "Visibilidade pública de atributo", correspondente a atributos definidos na área pública de uma classe.

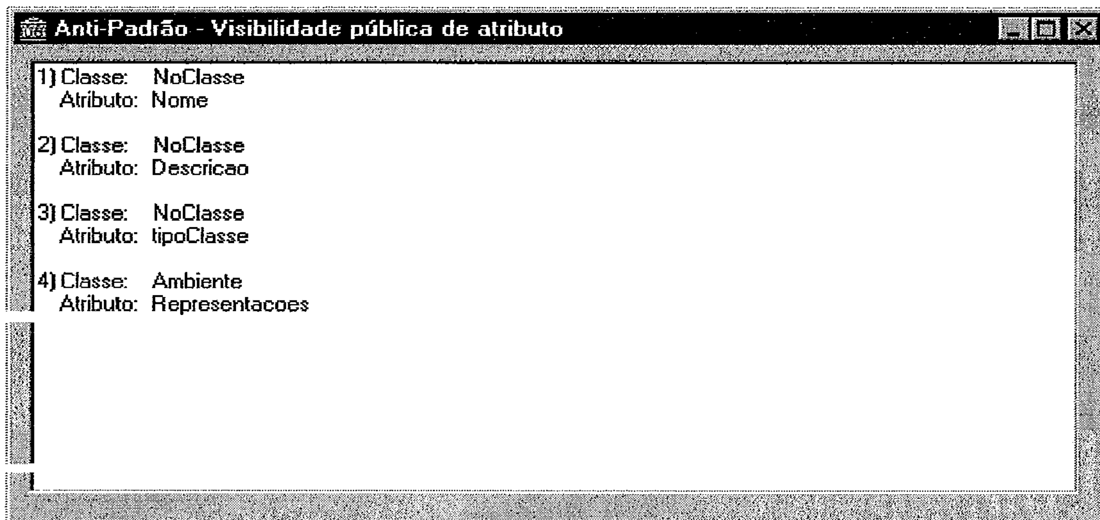


Figura 58: Resultado da pesquisa por um anti-padrão

O processo de detecção é realizado pela máquina de inferência do Visual Prolog. Para tanto, são utilizadas a base de fatos descrevendo o projeto, geradas conforme descrito na seção 4.4.5, e a base de regras para detecção de anti-padrões. A partir da seleção feita pelo usuário, são geradas consultas processadas pela máquina de inferência Prolog, gerando resultados que são exteriorizados na forma de números indicando a quantidade de anti-padrões detectados e também na forma de um relatório detalhando os elementos do modelo envolvidos no anti-padrão.

A detecção dos padrões é realizada de forma análoga aos anti-padrões. A opção de menu "*Detecção – Padrões*" dá acesso a uma janela de seleção de padrões, ilustrada na Figura 57. Após a execução do processo de detecção, cada padrão selecionado é atualizado com um número entre colchetes indicando o número de ocorrências de cada padrão detectado no modelo. Para visualizar os elementos envolvidos em cada ocorrência de um determinado padrão, basta selecionar o padrão através de um duplo clique, e a ferramenta apresenta uma janela com todas as ocorrências deste padrão com os respectivos elementos do modelo devidamente classificados, de acordo com os participantes do padrão. A Figura 59 ilustra o exemplo da ocorrência do padrão "*Abstract Factory*" (GAMMA et al., 1995).

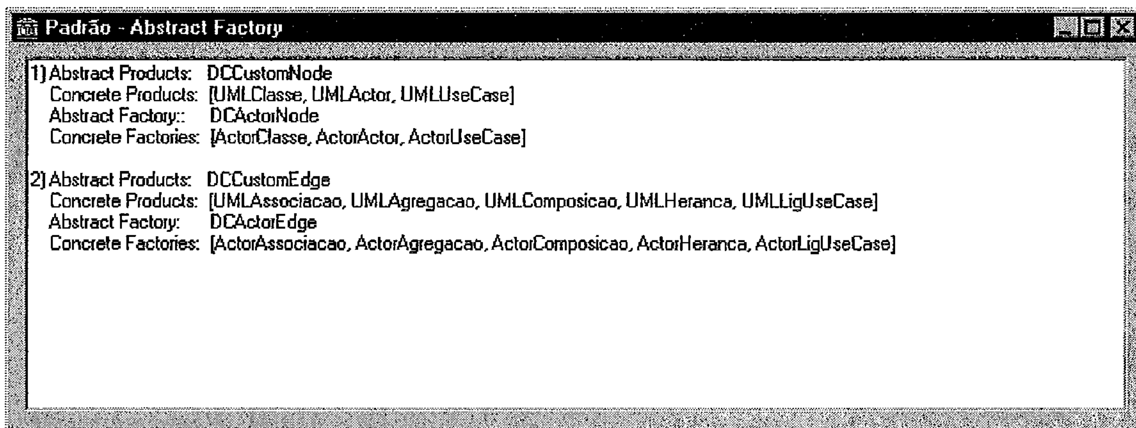


Figura 59: Resultado da pesquisa por um padrão

Desta forma, para efetuar a análise de um modelo de projeto orientado a objetos visando a detecção de padrões e anti-padrões, o projetista trabalha com a sua ferramenta CASE de projeto (Rose) e o OOPDTool. Na ferramenta CASE ele elabora o seu modelo de projeto. Em um dado momento, o projetista pode querer realizar a detecção de padrões/anti-padrões no modelo. Neste caso, ele utiliza o módulo *Analisador* do OOPDTool, conforme descrito anteriormente, que realiza a detecção com base nas informações presentes no modelo corrente na ferramenta CASE.

Suponha, por exemplo, que o projetista tenha escolhido detectar ocorrências do padrão *Factory Method* no seu modelo de projeto, ilustrado parcialmente na Figura 60 (diagrama de classes) e na Figura 61 (diagrama de colaboração ilustrando a instanciação da classe UMLUseCase realizada pelo método ActorUseCase::CreateNode).

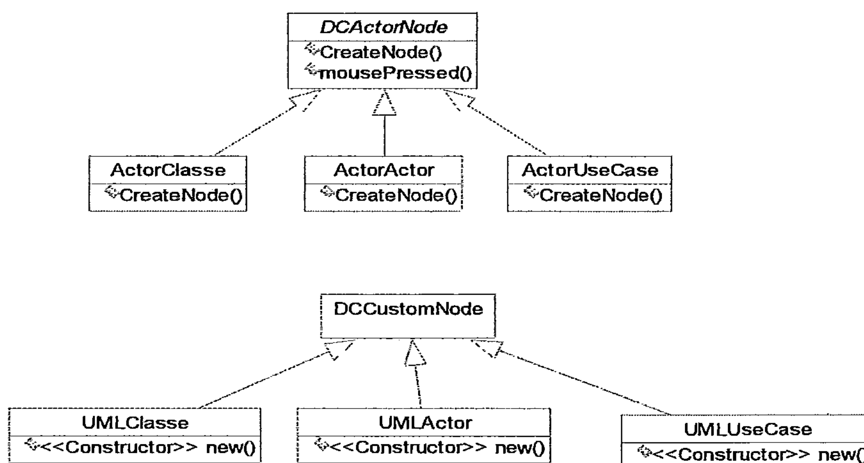


Figura 60: Diagrama de classes – exemplo

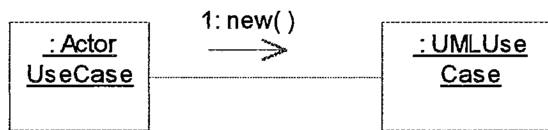


Figura 61: Diagrama de colaboração – exemplo

Cada padrão e anti-padrão está associado a uma regra em Prolog, que define um conjunto de condições que devem ser satisfeitas para que a construção seja detectada no modelo analisado. A regra associada à detecção do padrão *"Factory Method"* está armazenada na base de regras da forma ilustrada na Figura 62.

```
factoryMethodPattern(CreatorAbstrato, CreatorConcreto, ProdutoAbstrato, ProdutoConcreto, FactoryMethod) :-
        classificador(_, CreatorAbstrato, _, "Classe", "Abstrata", "NaoFolha", "Raiz"),
        operacao(CreatorAbstrato, FactoryMethod, FactoryMethodAbrev, "Instancia", _, _, "Virtual", "Abstrata", _),
        parametro(FactoryMethod, _, _, "Return", ProdutoAbstrato),
        operacao(CreatorAbstrato, Oper, _, "Instancia", _, _, _, "Metodo", _),
        invoca(Oper, _, FactoryMethod, "Self"),
        classificador(_, CreatorConcreto, _, "Classe", "Concreta", _, "NaoRaiz"),
        descendente(CreatorConcreto, CreatorAbstrato),
        operacao(CreatorConcreto,FactoryMethodRedefinido,FactoryMethodRedefAbrev,"Instancia",_,_,_, "Metodo",_),
        parametro(FactoryMethodRedefinido, _, _, "Return", ProdutoAbstrato),
        instancia(FactoryMethodRedefinido, ProdutoConcreto, _),
        descendente(ProdutoConcreto, ProdutoAbstrato),
        redefineOperacao(FactoryMethodRedefinido,FactoryMethod,FactoryMethodRedefAbrev, FactoryMethodAbrev),
        not (ProdutoConcreto = ProdutoAbstrato).
```

Figura 62: Regra para detecção do padrão *Factory Method*

Ao iniciar o processo de detecção, o OOPDTool gera uma base de fatos em Prolog correspondente aos elementos encontrados no modelo de projeto a ser analisado. Esta geração é realizada de acordo com o metamodelo definido no capítulo 3. A Figura 63 ilustra um extrato da base de fatos gerada para este modelo de exemplo.

O analisador passa o controle, então, para a máquina de inferência Prolog que, a partir da base de fatos gerada, verifica se as condições descritas no corpo da regra são satisfeitas por algum conjunto de elementos. Se estas condições forem satisfeitas, conclui-se que a construção (padrão/anti-padrão) está presente no modelo, e os termos do predicado cabeça da regra (neste exemplo seriam: *CreatorAbstrato, CreatorConcreto, ProdutoAbstrato, ProdutoConcreto* e *FactoryMethod*) irão corresponder aos nomes dos elementos do modelo cujas propriedades (descritas pelos fatos) satisfazem a regra (neste exemplo, *factoryMethodPattern*).

```
pacote("LogicalView","","Nil").
classificador("LogicalView","LogicalView::DCActor","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::DCActorNode","","Classe","Abstrata","NaoFolha","Raiz").
classificador("LogicalView","LogicalView::ActorClasse","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::ActorActor","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::ActorUseCase","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::DCCustomNode","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::UMLClasse","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::UMLActor","","Classe","Concreta","NaoFolha","NaoRaiz").
classificador("LogicalView","LogicalView::UMLUseCase","","Classe","Concreta","NaoFolha","NaoRaiz").
herdade("LogicalView::DCActorNode","LogicalView::DCActor").
herdade("LogicalView::ActorClasse","LogicalView::DCActorNode").
herdade("LogicalView::ActorActor","LogicalView::DCActorNode").
herdade("LogicalView::ActorUseCase","LogicalView::DCActorNode").
herdade("LogicalView::UMLClasse","LogicalView::DCCustomNode").
herdade("LogicalView::UMLActor","LogicalView::DCCustomNode").
herdade("LogicalView::UMLUseCase","LogicalView::DCCustomNode").
atributo("LogicalView::DCActorNode","LogicalView::DCActorNode::State","Instancia","Protected","int","int","NonConst","1","Composition")

atributo("LogicalView::DCActorNode","LogicalView::DCActorNode::SingleClick","Instancia","Protected","Boolean","Boolean","NonConst","
1","Composition").
atributo("LogicalView::DCActorNode","LogicalView::DCActorNode::StartPos","Instancia","Protected","LogicalView::Point","LogicalView::P
oint","NonConst","1","Association").
atributo("LogicalView::DCActorNode","LogicalView::DCActorNode::CurrentPos","Instancia","Protected","LogicalView::Point","LogicalView
::Point","NonConst","1","Association").
operacao("LogicalView::DCActor","LogicalView::DCActor::GetDiagramBox","GetDiagramBox","Instancia","Public","Read
Accessor","Virtual","Metodo","NonConst").
operacao("LogicalView::DCActorNode","LogicalView::DCActorNode::CreateNode","CreateNode","Instancia","Public","","Virtual","Abstrata
","NonConst").
operacao("LogicalView::DCActorNode","LogicalView::DCActorNode::mousePressed","mousePressed","Instancia","Public","","Virtual","Me
todo","NonConst").
operacao("LogicalView::ActorNodeVisao","LogicalView::ActorNodeVisao::SelecionaVisaoAbstrata","SelecionaVisaoAbstrata","Instancia",
"Public","","Virtual","Metodo","NonConst").
operacao("LogicalView::ActorClasse","LogicalView::ActorClasse::CreateNode","CreateNode","Instancia","Public","","Virtual","Metodo","N
onConst").
operacao("LogicalView::ActorActor","LogicalView::ActorActor::CreateNode","CreateNode","Instancia","Public","","Virtual","Metodo","NonC
onst").
operacao("LogicalView::ActorUseCase","LogicalView::ActorUseCase::CreateNode","CreateNode","Instancia","Public","","Virtual","Metodo
","NonConst").
operacao("LogicalView::UMLClasse","LogicalView::UMLClasse::new","new","Instancia","Public","Constructor","Virtual","Metodo","NonCo
nst").
operacao("LogicalView::UMLActor","LogicalView::UMLActor::new","new","Instancia","Public","Constructor","Virtual","Metodo","NonConst"
).
operacao("LogicalView::UMLUseCase","LogicalView::UMLUseCase::new","new","Instancia","Public","Constructor","Virtual","Metodo","No
nConst").
parametro("LogicalView::DCActor::GetDiagramBox","LogicalView::DCActor::GetDiagramBox::Return","1","Return","LogicalView::DCDiagr
amBox").
parametro("LogicalView::DCActorNode::CreateNode","LogicalView::DCActorNode::CreateNode::Return","1","Return","LogicalView::DCC
ustomNode").
parametro("LogicalView::ActorClasse::CreateNode","LogicalView::ActorClasse::CreateNode::Return","1","Return","LogicalView::DCCusto
mNode").
parametro("LogicalView::ActorActor::CreateNode","LogicalView::ActorActor::CreateNode::Return","1","Return","LogicalView::DCCustom
Node").
parametro("LogicalView::ActorUseCase::CreateNode","LogicalView::ActorUseCase::CreateNode::Return","1","Return","LogicalView::DC
CustomNode").
parametro("LogicalView::UMLClasse::new","LogicalView::UMLClasse::new::Return","1","Return","LogicalView::UMLClasse").
parametro("LogicalView::UMLActor::new","LogicalView::UMLActor::new::Return","1","Return","LogicalView::UMLActor").
parametro("LogicalView::UMLUseCase::new","LogicalView::UMLUseCase::new::Return","1","Return","LogicalView::UMLUseCase").
parametro("LogicalView::InstanciadorSemantico::instancia","LogicalView::InstanciadorSemantico::instancia::Return","1","Return","Logical
View::VisaoAbstrata").
instancia("LogicalView::ActorClasse::CreateNode","LogicalView::UMLClasse","LogicalView::UMLClasse::new").
instancia("LogicalView::ActorActor::CreateNode","LogicalView::UMLActor","LogicalView::UMLActor::new").
instancia("LogicalView::ActorUseCase::CreateNode","LogicalView::UMLUseCase","LogicalView::UMLUseCase::new").
invoca("LogicalView::DCActorNode::mousePressed","LogicalView::DCActorNode","LogicalView::DCActorNode::CreateNode","Self").
invoca("LogicalView::DCActorNode::mousePressed","LogicalView::DCDiagramBox","LogicalView::DCDiagramBox::AddNode","Atributo").
```

Figura 63: Extrato da Base de Fatos

Neste exemplo, a regra cuja cabeça corresponde ao predicado *factoryMethodPattern* é avaliada como verdadeira, quando instanciada com elementos da base de fatos conforme a Figura 64. Desta forma, os participantes do padrão definidos pelos termos *CreatorAbstrato, CreatorConcreto, ProdutoAbstrato, ProdutoConcreto* e *FactoryMethod,* correspondem, respectivamente, aos seguintes elementos do modelo: "LogicalView::DCActorNode", "LogicalView::ActorUseCase", "LogicalView::DCCustomNode","LogicalView::UMLUseCase","LogicalView::DCActorNode::CreateNode"

```
factoryMethodPattern("LogicalView::DCActorNode","LogicalView::ActorUseCase","LogicalView::DCCustomNode",
"LogicalView::UMLUseCase", "LogicalView::DCActorNode::CreateNode") :-

classificador("LogicalView","LogicalView::DCActorNode","","Classe","Abstrata","NaoFolha","Raiz").

operacao("LogicalView::DCActorNode","LogicalView::DCActorNode::CreateNode","CreateNode","Instancia","Public","","
Virtual","Abstrata","NonConst").

parametro("LogicalView::DCActorNode::CreateNode","LogicalView::DCActorNode::CreateNode::Return","1","Return","L
ogicalView::DCCustomNode").

operacao("LogicalView::DCActorNode","LogicalView::DCActorNode::mousePressed","mousePressed","Instancia","Publi
c","","Virtual","Metodo","NonConst").

invoca("LogicalView::DCActorNode::mousePressed","LogicalView::DCActorNode","LogicalView::DCActorNode::Create
Node","Self").

classificador("LogicalView","LogicalView::ActorUseCase","","Classe","Concreta","NaoFolha","NaoRaiz").

descendente("LogicalView::ActorUseCase","LogicalView::DCActorNode").

operacao("LogicalView::ActorUseCase","LogicalView::ActorUseCase::CreateNode","CreateNode","Instancia","Public",""
,"Virtual","Metodo","NonConst").

parametro("LogicalView::ActorUseCase::CreateNode","LogicalView::ActorUseCase::CreateNode::Return","1","Return","
LogicalView::DCCustomNode").

instancia("LogicalView::ActorUseCase::CreateNode","LogicalView::UMLUseCase","LogicalView::UMLUseCase::new").

descendente("LogicalView::UMLUseCase","LogicalView::DCCustomNode").

redefineOperacao("LogicalView::UMLUseCase::CreateNode", "LogicalView::DCActorNode::CreateNode, "CreateNode",
"CreateNode").

not (ProdutoConcreto = ProdutoAbstrato).
```

Figura 64: Exemplo de instanciação da regra "*factoryMethodPattern*"

4.7. Estudo de caso

A arquitetura proposta e a implementação do protótipo foram utilizados em um experimento prático utilizando dois projetos orientados a objetos no contexto do projeto Odyssey. Este experimento consistiu na detecção de padrões e anti-padrões nos projetos de duas ferramentas deste ambiente: PAC (ferramenta para configuração de processos de aquisição de conhecimento) e o editor de diagramas destinado à elaboração dos modelos definidos em (BRAGA e WERNER, 1999) cuja notação é baseada na UML (OMG, 1998). Estas duas ferramentas foram codificadas em aproximadamente 120 classes utilizando a linguagem Java.

O ponto de partida para a análise foi o código fonte destas ferramentas, e os modelos OO correspondentes. Os modelos disponíveis estavam em um nível de abstração alto, principalmente no que diz respeito à parte dinâmica de colaboração entre objetos, o que é justificável, uma vez que a geração de código realizada pela ferramenta CASE de modelagem utilizada no projeto faz uso apenas das informações estáticas, ou seja, aquelas presentes nos diagramas de classes. As informações estruturais sobre pacotes, classificadores, atributos e definição de operações foram, então, completadas com o auxílio do módulo de engenharia reversa da ferramenta CASE Rose, sendo que alguns ajustes foram manualmente realizados em função deste módulo não recuperar informações relativas às propriedades adicionadas pelo OOPDTool, conforme descrito na seção 4.4.5. A dinâmica de colaboração, correspondente à implementação dos métodos das classes, também foi adicionada manualmente, uma vez que a engenharia reversa do Rose captura apenas informações estáticas, ou seja, ela analisa apenas a definição das classes e não a implementação dos seus métodos.

A ênfase no processo de detecção de construções nos projetos analisados foi dada à detecção de anti-padrões, uma vez que a maior parte dos projetistas responsáveis não possuíam grande experiência em projetos OO, o que levava a crer que erros normalmente cometidos, especialmente por iniciantes, estariam presentes nestes projetos. A análise efetuada nestes projetos identificou vários anti-padrões, incluindo violações de heurísticas e algumas construções que poderiam ser substituídas pela aplicação de padrões. Um outro resultado interessante foi poder detectar a utilização de

um padrão em um contexto, onde na verdade o melhor teria sido a aplicação de um outro padrão. Segundo BROWN et al. (1998), uma causa freqüente de ocorrência de um anti-padrão é a aplicação de um padrão em um contexto incorreto, o que acabou sendo detectado pelo experimento. Alguns dos padrões e anti-padrões detectados no experimento são descritos nas próximas subseções.

4.7.1. Visibilidade Pública de Atributo

Este anti-padrão se caracteriza pela definição de um atributo na área pública de uma classe, correspondendo a uma possível forma de violação da heurística "*Atributos não devem ficar na área pública da classe*" (RIEL, 1996). Esta violação foi detectada em algumas classes do editor de diagramas. A Figura 65 mostra três atributos definidos na área pública da classe *NoClasse*, que corresponde a uma classe em um modelo orientado a objetos.

```
public class NoClasse extends ItemVisao
{
        public String    Nome;
        public String    Descricao;
        public int       tipoClasse;
}
```

Figura 65: Visibilidade pública de atributos

4.7.2. Visibilidade de atributo para subclasses

Este anti-padrão se caracteriza pela definição de um atributo na área "*protected*" de uma classe, correspondendo a uma forma de violação da heurística "*Todos os dados de uma classe base devem ser privados*" (RIEL, 1996). Esta violação foi detectada em algumas classes do editor de diagramas, como, por exemplo, nos atributos da classe ilustrada na Figura 66. A classe *DCActorNode* é uma classe abstrata que corresponde a um agente responsável pela inserção e posicionamento de nós em um diagrama. Estes nós podem corresponder a classes, casos de uso, ou seja, elementos de um diagrama. A partir desta classe, várias subclasses são definidas, cada uma especializada em um

determinado tipo de nó (elemento de um diagrama). Entretanto, a estrutura interna da classe *DCActorNode* está exposta para todas as subclasses, criando um acoplamento indesejável entre as mesmas.

```
abstract public class DCActorNode extends DCActor
{
        protected int           State = stIdle;
        protected Point         StartPos = new Point ();
        protected Point         CurrentPos = new Point ();
        protected boolean       SingleClick;
}
```

Figura 66: Visibilidade de atributo para subclasses

4.7.3. Herança de classe concreta

Este anti-padrão se caracteriza pela definição de um relacionamento de herança entre duas classes, onde a superclasse não é uma classe abstrata, o que caracteriza a violação da heurística "*Todas as classes base devem ser abstratas*" (MARTIN, 1995). Esta violação foi detectada em algumas hierarquias de classes do editor de diagramas, sendo que um caso mais grave ocorreu na hierarquia ilustrada na Figura 67, onde uma classe abstrata herda de uma classe concreta.

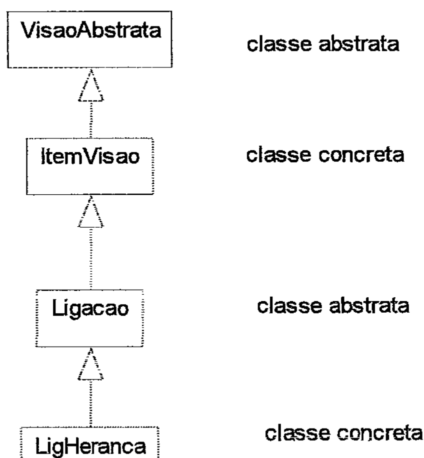


Figura 67: Herança de classe concreta

4.7.4. Classe "Entidade" definida a partir de uma classe utilitária

Este anti-padrão se caracteriza pela definição de uma relação de herança entre uma classe com estereótipo "Entidade" e uma classe com estereótipo "Coleção". Classes coleção correspondem àquelas destinadas ao armazenamento de vários objetos, como por exemplo, "*Vector*", "*List*", "*Queue*", entre outras. A definição de uma classe do domínio do problema (estereótipo "Entidade") a partir de uma herança de uma classe "Coleção" faz com que uma decisão de implementação (a estrutura de dados para armazenamento de uma coleção) fique exposta para os clientes, além de não poder ser modificada em tempo de execução, o que eventualmente pode ser necessário no caso de uma coleção (RIEL, 1996). A Figura 68 ilustra um exemplo capturado na ferramenta PAC, onde a classe *Característica* herda da classe *Vector*. Os estereótipos destas classes foram colocados manualmente, em função do processo de engenharia reversa do Rose não capturá-los automaticamente.

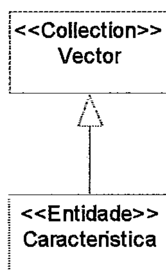


Figura 68: Classe "Entidade" definida a partir de uma classe utilitária

4.7.5. Exposição da implementação de uma coleção

Este anti-padrão consiste na definição de um método de acesso a um atributo correspondente a uma coleção de objetos, onde o tipo retornado por este método corresponde à classe coleção utilizada para definir o atributo. Normalmente, este anti-padrão surge em função do desconhecimento do padrão "*Iterator*" (GAMMA et al., 1995), que fornece uma solução para o problema de retornar uma coleção de objetos sem expor a estrutura de sua representação. A Figura 69 ilustra um exemplo deste anti-padrão detectado no editor de diagramas, onde a classe *NoClasse* expõe para seus

clientes a forma de armazenamento da coleção de *Atributos* e *Metodos*.

```
public class NoClasse extends ItemVisao
{
        private Vector          Atributos;
        private Vector          Metodos;

        public Vector PegaAtributos ()
        {
                return (Atributos);
        }

        public Vector PegaMetodos ()
        {
                return (Metodos);
        }
}
```

Figura 69: Exposição da implementação de uma coleção

4.7.6. Superclasse dependente de Subclasse

Este anti-padrão corresponde a uma violação da heurística "*Superclasses não devem conhecer nada a respeito de suas classes derivadas*" (RIEL, 1996). A Figura 70 captura um exemplo deste anti-padrão detectado no editor de diagramas. Neste exemplo, a superclasse *VisaoAbstrata* define a operação *PegaConjuntoVisoes* que retorna uma instância da subclasse *ConjuntoVisoes,* fazendo com que a superclasse *VisaoAbstrata* passe a depender da subclasse *ConjuntoVisoes*.

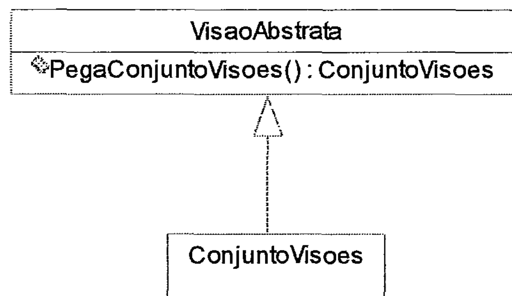


Figura 70: Superclasse dependente de Subclasse

4.7.7. Violação da Lei de Demeter

A lei de Demeter (LIEBERHERR, 1996) estabelece que os métodos de uma classe devem necessitar da colaboração apenas de objetos da própria classe, ou de atributos da classe (parentes próximos) ou de parâmetros (visitantes eventuais), com o objetivo de fazer com que cada método envie mensagens para objetos de um número limitado de outras classes. Uma violação comum desta lei é o encadeamento de chamadas para operações.

A Figura 71 ilustra um exemplo deste anti-padrão capturado no editor de diagramas, correspondendo a um caminho de chamadas que viola a Lei de Demeter. Este caminho de chamadas corresponde ao seguinte fragmento de programa em Java: *diagrama = GetDiagramBox().GetDiagram().GetSemantic();*

A resolução desta violação consiste na inclusão de uma operação *GetSemantic* na classe *DCDiagramBox*. Esta operação deve ser implementada através de uma delegação para a classe diretamente relacionada (*DCDiagram*), reduzindo o acoplamento entre as classes. Desta forma, a implementação da operação *GetSemantic* na classe *DCDiagramBox* deve ser *GetDiagram*().*GetSemantic*(), fazendo com que a classe *DCActor* passe a ficar dependente apenas da classe *DCDiagramBox*, uma vez que o trecho que violava a lei de Demeter passa a ser codificado em Java da seguinte forma: *diagrama = GetDiagramBox().GetSemantic();*

A violação da Lei de Demeter, normalmente, resulta na criação de programas com um número grande de interdependências entre as classes, podendo gerar sérios problemas de manutenção.

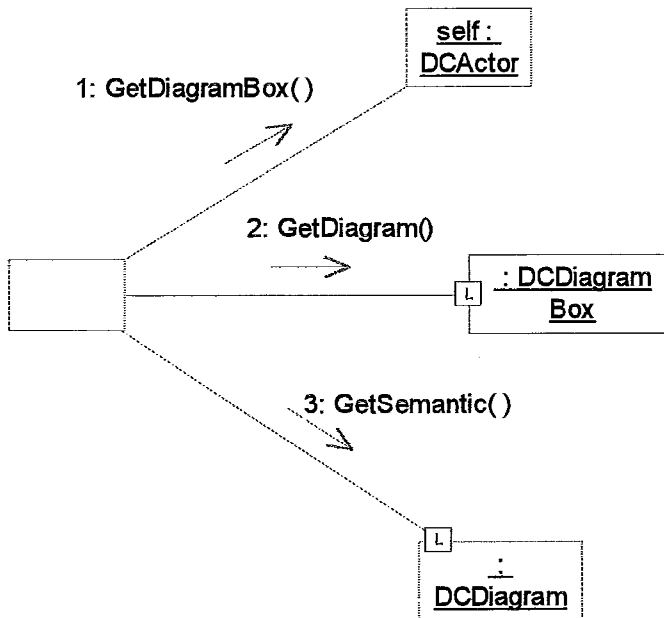


Figura 71: Violação da lei de Demeter.

4.7.8. Objeto com escopo global

Em determinadas situações de projeto, desejamos que uma classe tenha apenas uma instância e que esta instância seja facilmente acessível.. Uma solução de projeto frequentemente utilizada é definir esta instância ou como uma variável global, ou como um atributo público com escopo de classe, isto é, independente do número de instâncias desta classe, haverá somente um valor para este atributo. Esta solução se configura em um anti-padrão, caracterizando uma situação onde o padrão *"Singleton"* (GAMMA et al., 1995) deveria ter sido empregado.

Este anti-padrão foi detectado em algumas situações no editor de diagramas, como no exemplo ilustrado na Figura 72. Neste exemplo, a classe Ambiente define três atributos com escopo de classe na área pública. Em vários pontos da aplicação, o acesso, por exemplo, ao atributo *CriadorSemanticos*, que corresponde à implementação do padrão *"AbstractFactory"*, é realizado diretamente. Na definição do padrão *"AbstractFactory"* (GAMMA et al., 1995), a criação de sua única instância se dá através do padrão relacionado *"Singleton"*, o que não foi seguido nesta aplicação, dando origem a este anti-padrão.

```
public class Ambiente
{
        public static ListaDominios                    Dominios;
        public static ConjuntoRepresentacoes           Representacoes;
        public static FabricaSemanticos                CriadorSemanticos;
}
```

Figura 72: Objetos com escopo global

4.7.9. Factory method ao invés de Abstract Factory

Uma família de classes, normalmente utilizadas em conjunto, tem sua instanciação realizada em métodos nas subclasses clientes através da aplicação do padrão "*Factory Method*". As classes clientes formam uma hierarquia de classes, onde cada subclasse necessita trabalhar com uma subclasse da hierarquia de produtos. A aplicação do "*Factory Method*" permite criar métodos na raiz da hierarquia das classes clientes trabalhando de modo independente do tipo de produto, uma vez que a criação da instância específica de cada produto fica a cargo das subclasses clientes. Entretanto, esta solução torna os clientes dependentes de uma família específica de produtos. Uma solução mais flexível seria a aplicação do padrão "*Abstract Factory*" ao invés do "*Factory Method*". A Figura 73 ilustra um exemplo da ocorrência deste anti-padrão no contexto do editor de diagramas.

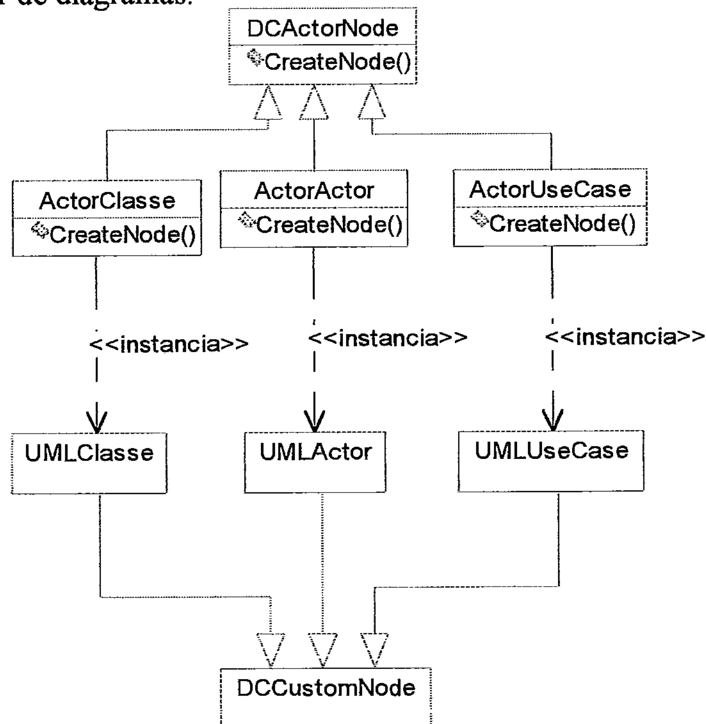


Figura 73: *Factory Method* ao invés de *Abstract Factory*

Neste exemplo, a hierarquia de classes clientes corresponde às subclasses de *DCActorNode*, responsáveis pela manipulação de produtos correspondentes às classes da hierarquia *DCCustomNode* (*UMLClasse, UMLActor, UMLUseCase*). A instanciação dos produtos é feita através do método *CreateNode*, resultante da aplicação do padrão "*Factory Method*". Cada produto corresponde a uma representação gráfica de classes, atores e casos de uso na notação proposta pela UML. Entretanto, a lógica das classes clientes não precisariam ficar dependentes de uma única forma de representação. Se o padrão "*AbstractFactory*" fosse aplicado, as subclasses clientes, ao invés de criarem diretamente as subclasses dos produtos, delegariam esta instanciação para uma classe *Factory* correspondente à notação desejada, permitindo que a hierarquia *DCActorNode* pudesse trabalhar com novas notações dos produtos sem precisar de alteração, seguindo o princípio Open/Closed (MEYER, 1997).

4.7.10. Factory method ao invés de Prototype

Este anti-padrão corresponde à aplicação do padrão "*Factory Method*", quando o mais indicado seria a aplicação do padrão "*Prototype*". Este anti-padrão, descrito implicitamente em (GAMMA et al., 1995), se caracteriza por uma proliferação de classes com a única responsabilidade de instanciar uma classe de uma hierarquia de produtos. A Figura 74 ilustra a ocorrência deste anti-padrão no contexto do editor de diagramas.

Neste exemplo, a classe *FábricaSemanticos* é responsável por encapsular a instanciação de uma família de produtos derivada da classe *VisaoAbstrata*. Esta instanciação é fornecida pelo método *Instancia* que retorna uma subclasse de *VisaoAbstrata*. Esta instanciação é realizada através de delegação para o método *Instancia* da subclasse *InstanciadorSemantico* correspondente ao *id* passado como parâmetro. Cada *InstanciadorSemantico* é registrado na inicialização da aplicação através do método *Registra* da classe *FabricaSemanticos*. Para cada novo produto, isto é, subclasse de *VisaoAbstrata*, é necessário criar uma subclasse de *InstanciadorSemantico* com a única responsabilidade de criar o subproduto correspondente. Isto faz com que ocorra uma proliferação desnecessária de classes, criando um outro ponto de evolução com o qual o projetista tem que se preocupar, ou seja, a criação de um novo produto exige a expansão de duas hierarquias de classes: uma correspondente aos produtos e a outra correspondente aos instanciadores.

A solução mais adequada seria a aplicação do padrão *"Prototype"*, onde cada classe da hierarquia de produtos define um método *Clone* com a responsabilidade de instanciar um outro objeto de sua própria classe, gerando um clone. A classe *FábricaSemanticos* passaria a registrar uma instância de cada produto (protótipos) e não mais instanciadores. A implementação do método *Instancia* da *FabricaSemanticos* passaria a chamar o método *Clone* correspondente ao protótipo do produto desejado, eliminando a necessidade de criar a hierarquia de classes de instanciação (*InstanciadorSemantico*), dando a mesma flexibilidade e independência do processo de instanciação que era obtido com o padrão *"FactoryMethod"*, mas com uma grande redução do número de classes necessárias para a sua implementação.

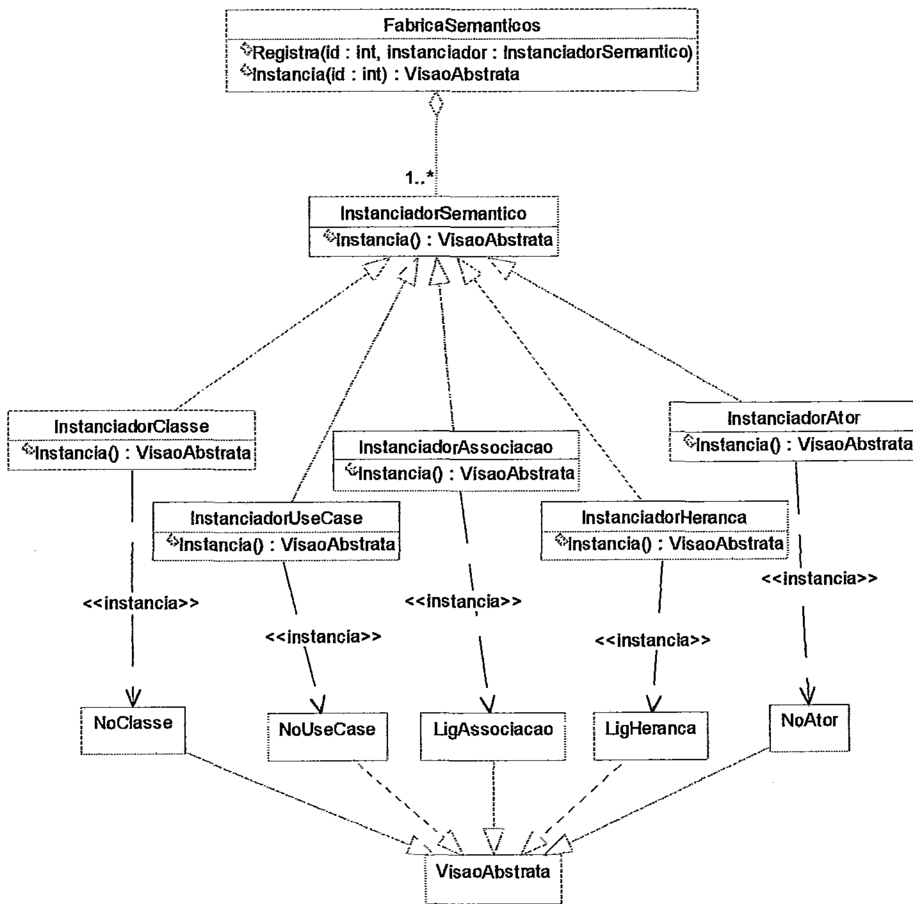


Figura 74: *Factory method* ao invés de *Prototype*

4.8. Conclusões

Este capítulo apresentou a implementação de um protótipo correspondente à parte da arquitetura proposta para a integração de heurísticas, padrões e anti-padrões em ferramentas CASE de projetos orientados a objetos. Um estudo de caso foi realizado, mostrando a utilidade desta arquitetura no sentido de obter projetos orientados a objetos mais flexíveis e reutilizáveis. Vários problemas detectáveis ainda em estágio de projeto podem ser identificados de forma automática. A existência de um repositório onde estes problemas possam ser catalogados e disponibilizados para os projetistas permite, não somente um suporte para a sua detecção, mas também para a sua divulgação e conseqüente disseminação deste conhecimento.

Cada padrão/anti-padrão deve ter suas respectivas regras de detecção perfeitamente definidas para que a ferramenta possa realizar o processo de detecção. Esta regra deve ser definida pelo especialista que é responsável por definir todas as condições que devem ser satisfeitas para que se possa concluir a presença de um determinado padrão/anti-padrão. Através deste experimento, foi possível verificar que padrões complexos, como o *Factory Method* (GAMMA et al, 1995), por exemplo, são difíceis de serem formalizados, exigindo um processo de depuração da(s) regra(s) associada(s). Neste experimento, verificamos a ocorrência de falsos positivos (indicação de um padrão quando na realidade ele não ocorria no modelo analisado) e de falsos negativos (padrões que ocorriam no modelo mas não eram indicados pela ferramenta). Estes falsos positivos e negativos foram ocasionados pela definição incorreta das regras de detecção, o que gerou a necessidade de modificações nas regras até que estas ocorrências fossem eliminadas.

Neste experimento, os modelos foram manualmente analisados em busca de todas as ocorrências dos padrões e anti-padrões catalogados. Os resultados gerados pela detecção automática realizada pelo OOPDTool foram confrontados com os resultados desta análise manual, permitindo identificar a ocorrência de falsos positivos e falsos negativos, e refinar a definição das regras de detecção destas construções.

A divulgação dos resultados deste experimento, especialmente as construções ruins detectadas, permitiu que os projetistas rapidamente absorvessem este conhecimento, o que possibilitou a geração de uma segunda versão do editor de

diagramas, já incorporando as transformações propostas pelos anti-padrões, resultando em projeto com um maior grau de flexibilidade e qualidade.

Este experimento mostrou também que o conhecimento de linguagens orientadas a objetos e de métodos de desenvolvimento OO, assim como a utilização de ferramentas CASE atualmente disponíveis para a elaboração de projetos OO, não são suficientes para a produção de projetos flexíveis e reutilizáveis. A maior parte dos projetistas detinham, de alguma forma, este conhecimento, o que não impediu o aparecimento de diversos anti-padrões. A proposta de integrar heurísticas, padrões e anti-padrões a este contexto visa preencher esta lacuna.

Capítulo 5

Conclusões

Desde o final da década passada, inúmeras organizações vêm empregando o paradigma de orientação a objetos no desenvolvimento de software. A crescente popularidade deste novo paradigma pode ser creditada à divulgação dos resultados que potencialmente podem ser atingidos com o seu uso, especialmente em relação ao aumento de produtividade no desenvolvimento de novos sistemas, e à flexibilidade dos sistemas produzidos, possibilitando uma dramática redução no esforço de manutenção.

Entretanto, em várias destas organizações os benefícios percebidos ficaram aquém das expectativas, com a produção de sistemas inflexíveis, hoje denominados sistemas OO legados, mesmo com a utilização de métodos e ferramentas CASE de suporte ao desenvolvimento segundo este paradigma.

Neste contexto, esta tese buscou realizar um estudo sobre formas de conhecimento que pudessem auxiliar os projetistas de software orientado a objetos na geração de projetos mais flexíveis e reutilizáveis, tentando maximizar os benefícios potencialmente atingíveis através da utilização do paradigma OO. Foi proposta uma arquitetura para a integração deste conhecimento, na forma de heurísticas, padrões e anti-padrões, a ferramentas CASE normalmente utilizadas no desenvolvimento de sistemas OO.

Esta integração disponibiliza uma base de heurísticas, padrões e anti-padrões de projeto, de forma organizada e padronizada, tornando explícito os relacionamentos porventura existentes entre estes conceitos. Esta base possibilita a disseminação de um conhecimento hoje disperso e, muitas vezes, presente de forma implícita na literatura. Além disso, possibilita o registro e a disseminação de conhecimento resultante de experiências dos próprios membros de uma organização em soluções de projeto, tanto as boas como as ruins. Embora os benefícios resultantes do emprego de padrões de projeto ainda não tenham sido objetivamente quantificados através de métricas e processos de validação científicos, a disponibilização e emprego deste conhecimento têm-se mostrado de grande valia para a produção de sistemas mais flexíveis e

reutilizáveis. Cabe aos responsáveis pela alimentação desta base de conhecimento julgar quais padrões e anti-padrões são importantes e devam ser colocados nesta base, uma vez que a proliferação descontrolada destas construções pode gerar um efeito negativo do ponto de vista de sua busca e utilização.

Esta base de conhecimento possibilita, também, a detecção de construções típicas em projetos orientados a objetos. Estas construções podem corresponder tanto a boas soluções (padrões de projeto) como ruins, que potencialmente levam a problemas de manutenção e reutilização futura (anti-padrões). A possibilidade de detectar anti-padrões ainda em tempo de projeto faz com que seja possível substituí-las antes que estejam codificadas, ou até mesmo já em produção, representando uma potencial redução no custo de manutenção.

Por fim, um protótipo de parte da arquitetura proposta (OOPDTool) foi desenvolvido e um estudo de caso realizado no contexto do projeto Odyssey em desenvolvimento na COPPE/UFRJ, ilustraram o ganho potencial de produtividade, qualidade e disseminação de conhecimento que podem ser obtidos.

Dentre os diversos tópicos abordados neste trabalho, destacam-se as seguintes contribuições desta tese:

- Detecção de fontes de conhecimento, não enfatizadas pelos métodos de desenvolvimento orientado a objetos, importantes para que um projetista consiga gerar projetos orientados a objetos mais flexíveis e reutilizáveis.

- Detecção de relacionamentos entre heurísticas, padrões e anti-padrões, conceitos apresentados de forma separada na literatura técnica de orientação a objetos, e a proposta de tratá-los de forma integrada em um ambiente de desenvolvimento.

- Elaboração de uma abordagem de integração destes conceitos à ferramentas CASE de desenvolvimento orientado a objetos, fornecendo, ainda, suporte automatizado para a detecção de padrões e, especialmente, de anti-padrões.

- Realização de um experimento de utilização da abordagem no contexto do projeto Odyssey. O experimento, apesar de não consistir em uma validação

completa das ferramentas em desenvolvimento no contexto do projeto, possibilitou visualizar a utilidade da proposta dentro de um ambiente típico de desenvolvimento OO, com vários desenvolvedores com um bom conhecimento dos conceitos básicos de orientação a objetos, mas sem grande experiência em desenvolvimento de sistemas grandes e complexos. Várias construções ruins (anti-padrões) foram identificadas, e a divulgação destes resultados permitiu que os projetistas rapidamente absorvessem este conhecimento. Isto contribuiu para que a geração de novas versões das ferramentas incorporasse as transformações propostas pelos anti-padrões, assim como evitou que estas construções se alastrassem nos novos módulos, resultando em um projeto com um maior grau de flexibilidade e qualidade.

Este trabalho se concentrou em características não encontradas em outras propostas, tais como a administração de heurísticas e anti-padrões, a integração destes conceitos com o conceito de padrões, a ampliação do suporte à detecção de padrões através da extração de informações sobre a parte dinâmica de um projeto orientado a objetos e, ainda, um suporte automatizado à detecção de anti-padrões, cuja utilidade foi demonstrada pelo experimento.

Dentre as perspectivas futuras à continuidade e aperfeiçoamento deste trabalho, destacam-se as seguintes:

a) a catalogação extensiva de heurísticas, padrões e anti-padrões, e a realização de experimentos de validação mais abrangentes em outros projetos de desenvolvimento orientado a objetos, podendo-se, inclusive, obter indicadores, através de grupos de controle, sobre a influência que este conhecimento pode exercer na qualidade dos projetos desenvolvidos com e sem este conhecimento. Outras medidas que poderiam ser obtidas com estes experimentos em maior escala são aquelas relacionadas à acurácia da detecção das construções, tais como a incidência de falso positivos e negativos, por exemplo.

b) ampliação do suporte à utilização destes conceitos, incorporando à abordagem funcionalidades tais como:

- auxílio ao desenvolvedor na busca e seleção das heurísticas e padrões mais adequados para um problema em particular. Esta assistência pode ser fornecida

com o auxílio de técnicas de Inteligência Artificial, como por exemplo, através do desenvolvimento de sistemas especialistas, ou ainda, da utilização de mecanismos de aprendizado de máquina, como redes neurais e programação lógica indutiva (MITCHELL, 1997). Hoje esta assistência está limitada a mecanismos de navegação e consulta.

- suporte para a captura das regras que definem os padrões e anti-padrões através de um processo de treinamento, utilizando técnicas de aprendizado de máquina. Hoje, o especialista tem que definir exatamente as características que estabelecem de modo não ambíguo uma construção (padrão ou anti-padrão), representando-as em Prolog na ferramenta OOPDTool, embora a arquitetura proposta não imponha que esta representação seja feita utilizando Prolog. A idéia é que o especialista apresente exemplos característicos de uma determinada construção, e a ferramenta possa aprender ou refinar uma regra que caracterize esta construção. Este treinamento pode ser baseado em técnicas de programação lógica indutiva, por exemplo.
- suporte para a detecção de padrões e anti-padrões através de semelhança. Hoje o processo de detecção está limitado a uma correspondência exata entre fatos de um projeto e regras que definem uma construção típica. A idéia é possibilitar a detecção de padrões e anti-padrões, mesmo que estes ocorram em construções parecidas, correspondentes a pequenas variações da definição típica de uma construção. Novamente, o emprego de técnicas de aprendizado de máquina fornecem um caminho promissor (RAMON et al., 1998).
- suporte para a reestruturação automática ou semi-automática de um projeto, de modo a transformar construções ruins eventualmente detectadas (anti-padrões), em construções mais adequadas. Esta reestruturação pode ser realizada de forma automática em casos simples, como por exemplo, no anti-padrão *Visibilidade Pública de Atributo,* podendo-se empregar técnicas como as descritas em (OPDYKE, 1992) que definem operações de reestruturação aplicáveis a projetos orientados a objetos, sem alterar a sua funcionalidade. Entretanto, anti-padrões como o *"Blob"*, por exemplo, exigem um conhecimento semântico do modelo, o que, provavelmente, implica na intervenção do projetista na reestruturação, daí a necessidade de um suporte semi-automático. Esta reestruturação poderia, inclusive, automatizar o processo de substituição de anti-padrões por padrões, nas situações onde isso fosse adequado e possível.

- reconhecimento de construções típicas, mesmo que estas não tenham sido explicitamente catalogadas na base de conhecimento. As construções identificadas seriam, então, investigadas por um especialista, podendo corresponder a um padrão ou um anti-padrão que deva ser catalogado na base. Hoje, a abordagem está limitada à deteção de construções catalogadas na base de padrões e anti-padrões.
- suporte mais sofisticado para a visualização dos padrões utilizados em um determinado sistema, o que exige um estudo mais aprofundado, uma vez que um modelo de projeto orientado a objetos tipicamente é composto por dezenas de diagramas, tanto de classe como de colaboração, ou seja, existem múltiplas formas de visualizar os componentes de um projeto OO. Hoje, a ferramenta apresenta o resultado da detecção apenas de forma textual.
- suporte para a instanciação de padrões em um projeto. Como algumas propostas já fornecem este tipo de suporte, conforme ilustrado pela tabela 2, decidiu-se não colocar este suporte na primeira versão da OOPDTool, uma vez que a ênfase do trabalho foi dada às características ausentes em outras abordagens.
- coleta de métricas sobre a utilização dos diversos padrões e anti-padrões, permitindo identificar, por exemplo, as construções mais utilizadas, aquelas menos ou não utilizadas, ou até mesmo padrões que necessitem ser melhorados em função de suas utilizações necessitarem de um nível de adaptação acima do normal.
- integração de padrões e anti-padrões nos diversos níveis de abstração (análise, arquitetura, projeto e idiomas), avaliando-se como seria possível fornecer um suporte automatizado para a incorporação de padrões à medida em que passamos de um nível para outro.
- integração com outras abordagens de qualidade de software, como por exemplo, a área de métricas de projeto orientado a objetos. A utilização combinada de métricas, heurísticas, padrões e anti-padrões pode possibilitar não somente a detecção de pontos do projeto que possam ser melhorados, mas também um suporte na avaliação de qual das possíveis transformações aplicáveis em uma situação particular é a mais indicada.

A Tabela 2 resgata o quadro com o resumo de diversos trabalhos nesta área, descritos no capítulo 2 (Tabela 1), com a inclusão da arquitetura apresentada nesta tese (OOPDTool).

	PAT	*KT*	*OMT Tool*	*Depars*	*DPT*	*IBM*	*Waterloo*	*Florijn*	*Fram. Studio*	*OOPD Tool*
Administração de um repositório de heurísticas										✓
Administração de um repositório de padrões						somente consulta		✓	✓	✓
Administração de um repositório de anti-padrões										✓
Integração dos conceitos de heurísticas, padrões e anti-padrões										✓
Detecção de padrões estruturais	✓	✓		✓						✓
Detecção de padrões comportamen-tais e de instanciação										✓
Detecção de anti-padrões										✓
Instanciação de padrões em projeto							✓	✓	✓	
Instanciação de padrões em código						✓	✓			
Suporte à reestruturação de projeto baseada em padrões			✓		✓					

Tabela 2: Quadro resumo atualizado

REFERÊNCIAS BIBLIOGRÁFICAS

(ABREU, 1998). F. B. Abreu, "Reengineering the Modularity of Object Oriented Systems", Em *Workshop on Techniques, Tools and Formalisms for Capturing and Assessing the Architectural Quality in Object Oriented Software*, ECOOP 98, Bruxelas, Bélgica, Julho, 1998.

(AKROYD, 1996). M. Akroyd, "AntiPatterns Session Notes". Em *Object World West*, São Francisco, Estados Unidos, 1996.

(ALENCAR et al., 1995). P.S.C. Alencar, D.D. Cowan, K.J.Lichtner, C.J.P. Lucena, L.C.M.Nova, "Tool Support for Formal Design Patterns". Em *Tech. Rep. CS-95-36*, Universidade de Waterloo, Ontario, Canadá, Agosto, 1995.

(ALEXANDER 1979). C. Alexander, *The Timeless Way of Building*, 1 ed., New York, Estados Unidos, Oxford University Press, 1979.

(ALEXANDER et al., 1977). C. Alexander, S. Ishikawa, M.Silverstein, M. Jacobson, I. King-Fiksdahl, S. Angel, *A Pattern Language*, 1 ed., New York, Estados Unidos, Oxford University Press, 1977.

(AMBLER, 1998). S. Ambler, *Building Object Applications That Work*, 1 ed., Cambridge University Press, 1998.

(APPLETON, 1997). B. Appleton, *Patterns and Software: Essential Concepts and Terminology*, Internet: http://www.enteract.com/~bradapp/docs/patterns-intro.html>, 1997.

(BLUEPRINT, 1999). Blueprint Technologies, *Framework Studio User Guide*, Canadá, 1999.

(BOOCH et al., 1999). G. Booch, I. Jacobson, J. Rumbaugh, *The Unified Modeling Language User Guide*, 1 ed., Reading, Massachusetts, Addison-Wesley, 1999.

(BOOCH, 1994). G. Booch, *Object Oriented Analysis and Design with Applications*, 2 ed., Redwood City, California, The Benjamin/Cummings Publishing Company, 1994.

(BRAGA e WERNER, 1999). R. Braga, C. Werner. "Odyssey-DE: Um Processo para Desenvolvimento de Componentes Reutilizáveis", Em *X Conferência Internacional de Tecnologia de Software*, Curitiba, Maio, 1999.

(BRAGA et al., 1998). R. Braga, C. Werner, M. Mattoso. "A Reuse Infrastructure Based on Domain Models", Em *Proc. of ICCT'98*, Canadá, Junho, 1998.

(BRAGA et al., 1999). R. Braga, C. Werner, M. Mattoso. "Odyssey: A Reuse Environment Based on Domain Models", Em *Proc. of 2nd. IEEE Symposium on Application-Specific Systems and Software Engineering Technology,* Richardson, Estados Unidos, Março, 1999.

(BROCKSCHMIDT, 1995). K. Brockschmidt, *Inside OLE,* 2 ed., Redmond, Washington, Microsoft Press, 1995.

(BROWN et al., 1998). W. Brown, R. Malveau, H. McCormick III, T. Mowbray. *Anti-patterns – Refactoring Software, Architectures, and Projects in Crisis,* 1 ed., Wiley Computer Publishing, 1998.

(BROWN, 1997). K. Brown, *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk,* Tese de Mestrado.
Internet: http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm

(BUDINSKY et al., 1996). F.J. Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu, "Automatic code generation from design patterns", Em *IBM Systems Journal,* Vol. 35, No. 2, 1996.

(BUSCHMANN et al., 1996). F.Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns,* 1 ed., Chichester, Reino Unido, John Wiley & Sons, 1996.

(CHIDAMBER e KEMERER, 1994). S. Chidamber, C. Kemerer, "A Metrics Suite for Object Oriented Design", Em *IEEE Transactions on Software Engineering,* v. 20, n.6, Junho, 1994.

(CINNÉIDE e NIXON, 1998). M. Cinnéide, P. Nixon, "Program Restructuring to Introduce Design Patterns". Em *Proc. of the Workshop on Object Oriented Software Evolution and Re-Engineering*, ECOOP, Bruxelas, Bélgica, Julho, 1998.

(CIUPKE, 1998). O. Ciupke, "Analysis of Object Oriented Programs Using Graphs", Em *Proc. of the Workshop on Object Oriented Software Evolution and Re-Engineering*, ECOOP, Bruxelas, Bélgica, Julho, 1998.

(CLUNIE, 1997). C. Clunie, *Avaliação de Qualidade em Especificações Orientadas a Objetos*, Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 1997.

(COAD e YOURDON, 1991). P. Coad, E. Yourdon, *Object Oriented Analysis*, 2 ed., Englewood Cliffs, N. J., Yourdon Press, 1991.

(COAD, 1992). P. Coad, "Object oriented patterns". Em *Communications of the ACM*, 35(9), Setembro 1992.

(COLEMAN et al., 1994). D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeramaes, *Object Oriented Development: The Fusion Method*, Englewood Cliffs, N.J., Prentice Hall, 1994.

(COPLIEN e SCHMIDT, 1995). J. O. Coplien, D. Shcmidt, *Pattern Languages of Program Design*, 1 ed., Addison-Wesley, 1995.

(COPLIEN, 1996). J. O. Coplien, *Software Patterns*, 1 ed., SIGS Books, 1996.

(COPLIEN, 1997). J.O.Coplien, *AntiPatterns page*. Internet: http://c2.com/cgi/wiki?AntiPatterns

(D'SOUZA e WILLS, 1999). D. D'Souza, A. Wills, *Object, Components and Frameworks with UML. The Catalysis Approach*, 1 ed., Reading, Massachusetts, Addison-Wesley, 1999.

(DEMEYER et al., 1998). S. Demeyer, S. Tichelaar, P. Steyaert. *FAMOOS – Definition of a Common Exchange Model*. http://www.iam.unibe.ch/~famoos/InfoExchFormat/

(FLORIJN et al., 1997). G. Florijn, M. Meijers, P. van Winsen. "Tool support for object oriented patterns". Em *European Conference on Object Oriented Programming*. Springer-Verlag, 1997.

(GALL et al., 1996). H. Gall, R. Klosch, R. Mittermeir, "Application Patterns in Re-Engineering: Identifying and Using Reusable Concepts". Em *6th. International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems (IPMU' 96)*, Julho, 1996.

(GAMMA et al., 1995). E. Gamma, R.Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, 1 ed., Reading, Massachusetts, Addison-Wesley, 1995.

(HENDERSON-SELLERS, 1996). B. Henderson-Sellers, *Object Oriented Metrics*, 1 ed., N. J., Prentice-Hall, 1996.

(JACOBSON et al, 1999). I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, 1 ed., Reading, Massachusetts, Addison-Wesley, 1999.

(JACOBSON et al., 1992). I. Jacobson, M Christerson, P. Jonsson, G. Overgaard, *Object Oriented Software Engineering*, 1 ed., Workingham, Inglaterra, Addison-Wesley, 1992.

(JOHNSON e FOOTE, 1988). R. Johnson and B. Foote, "Designing reusable classes". Em *Journal of OO Programming*, SIGS Publications, 1(2): 22-35, Junho 1988.

(JOHNSON, 1995). J. Johnson, "Creating Chaos". Em *American Programmer*, Julho, 1995.

(KELLER et al., 1999), R. Keller, R. Schauer, S. Robitaille, P. Pagé, "Pattern-Based Reverse-Engineering of Design Components", Em *International Conference on Software Engineering – ICSE'99*, pp. 226-235, Los Angeles, CA, 1999.

(KOENIG, 1995). A. Koenig. "Patterns and antipatterns". Em *Journal of Object Oriented Programming*, 8(1), Março, 1995.

(KRAMER e PRECHELT, 1996). C. Kramer, L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software". Em *Proc. Working Conf. On Reverse Engineering*, IEEE CS press, Monterey, Novembro, 1996.

(LEA, 1994). D. Lea, "Christopher Alexander: An Introduction for Object-Oriented Designers". Em *ACM SIGSOFT Software Engineering Notes* 19, 1, pp. 39-46, 1994.

(LIEBERHERR, 1996). K. J. Lieberherr. *Adaptive Object Oriented Software. The Demeter method with propagation patterns.* 1 ed., PWS Publishing Company, 1996.

(MARTIN et al., 1997). R. Martin, D. Riehle, F. Buschmann. *Pattern Languages of Program Design 3*, 1 ed., Addison Wesley, 1998.

(MARTIN, 1995). R. Martin, *Designing Object Oriented C++ Applications using the Booch Method*, 1 ed., Uper Saddle River, N. J., Prentice Hall, 1995.

(MARTIN, 1996a). R. Martin, "The Open-Close Principle", Em *C++ Report*, Janeiro, 1996.

(MARTIN, 1996b). R. Martin, "The Dependency Inversion Principle", Em *C++ Report*, Maio, 1996.

(MARTIN, 1996c). R. Martin, "The Interface Segregation Principle", Em *C++ Report*, Agosto, 1996.

(MAUGHAN e AVOTINS, 1998). G. Maughan, J. Avotins. "A Meta-model for Object Oriented Reengineering and Metrics Collection". Em *Eiffel Liberty Journal*. Vol. 1, No. 4., 1998. URL: http://www.elj.com/elj/v1/

(MEYER, 1997). B. Meyer, *Object Oriented Software Construction*, 2 ed., Cambridge, Prentice-Hall, 1997.

(MITCHELL, 1997). T. M. Mittchel. *Machine Learning*. 1 ed., McGraw Hill, 1997.

(MOWBRAY e MALVEAU, 1997). T. Mowbray, R. Malveau, *CORBA Design Patterns*, 1 ed., Wiley Computer Publishing, 1997.

(OMG, 1998). Object Management Group, *Unified Modeling Language Specification,* Framingham, Massachusetts, 1998. Internet: www.omg.org

(OPDYKE, 1992). W. F. Opdyke. *Refactoring object oriented frameworks.* Tese de Doutorado, Universidade de Illinois, Urbana, Illinois, 1992.

(PAGEL e WINTER, 1996). B. Pagel, M. Winter, *Towards Pattern-Based Tools,* Technical Report, Universidade de Hagen, 1996.

(PARNAS, 1979) D. Parnas. "On the criteria to be used in decomposing systems into modules". Em *Classics in Software Enginerring.* Yourdon Press. New York, NY, 1979.

(PLATINUM, 1999). Platinum Technologies, Computer Associates, Inc. *Paradigm Plus CASE Tool.* Internet: http://www.platinum.com/products/brochure/als/pplus/pp_il2.htm

(PRESSMAN, 1997). R.S. Pressman, *Software Engineering, A practitioner"s approach,* 4 ed., McGraw Hill, 1997.

(RAMON e BRUYNOOGHE, 1998). J. Ramon e M. Bruynooghe. "A framework for defining distances between first-order logic objects". Em *Proc. of the 8th International Conference on Inductive Logic Programming. Lecture Notes in Artificial Intelligence,* pp. 271-280. Springer-Verlag, 1998.

(RAMON et al., 1998). J. Ramon, M. Bruynooghe e W. Van Laer. "Distance measures between atoms". Em *Proceedings of the CompulogNet Area Meeting on "Computational Logic and Machine Learning",* pp. 35-41, 1998.

(RATIONAL, 1998). Rational Software Corporation, *Rational Rose 98 Extensibility User Guide,* Publications Department, 2800 San Tomas Expressway, Santa Clara, CA, 1998.

(RIEHLE e ZULLIGHOVEN, 1996). D. Riehle e H. Zullighoven. "Understanding and Using Patterns in Software Development". Em *Theory and Practice of Object Systems,* vol. 1, 1996.

(RIEL, 1996). A. Riel, *Object Oriented Design Heuristics,* 1 ed., Addison-Wesley, 1996.

(ROSETI et al., 1998). M. Roseti, L. Murta, C. Werner, "Uma Ferramenta para Configuração do Processo de Aquisição de Conhecimento no Contexto de Análise de Domínio". *Em XII Simpósio Brasileiro de Engenharia de Software. 1a. Mostra Brasileira de Software Acadêmico e Comercial*, Maringá, Outubro, 1998.

(ROSETI, 1998). M.Z. Roseti. *Uma Proposta de Sistemática para Aquisição de Conhecimento no Contexto de Análise de Domínio*, Tese de Mestrado, COPPE/UFRJ, novembro, 1998.

(RUMBAUGH et al., 1991). J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object Oriented Modeling and Design*, 1 ed., Englewood Cliffs, N.J., Prentice Hall, 1991.

(RUMBAUGH et al., 1999). J. Rumbaugh, G. Booch, I. Jacobson, *The Unified Modeling Language Reference Manual*, 1 ed., Reading, Massachusetts, Addison-Wesley, 1999.

(SALVIANO 97). C. Salviano, "Introdução a Software Patterns", Em *XI Simpósio Brasileiro de Engenharia de Software (SBES) - Tutorial*, Fortaleza, 1997.

(SCG, 1998). Software Composition Group (SCG), *FAMOOS Project*, Forshunngszentrum Informatik, Internet: http: //iamwwww.unibe.ch/~famoos.

(SCHMIDT et al., 1996). D. Schmidt, R. Johnson, M. Fayad, "Software Patterns". Em *Communications of the ACM, Special Issue on Patterns and Pattern Languages*, Vol. 39, No. 10, Outubro, 1996.

(SCHULZ, 1998). B. Schulz, "Design Patterns as Operators Implemented with Refactoring", Em *Workshop on Experiences in Object Oriented Re-Engineering*, ECOOP, Bruxelas, Bélgica, Julho, 1998.

(SELECT, 1999). Select Software Tools, Inc., *Select Enterprise CASE Tool*. Internet: http://www.select-software.com.

(SHAW, 1989). M. Shaw, "Larger Scale Systems Require Higher-Level Abstractions", Em *Proceedings of Fifth International Workshop on Software Specification and Design*. IEEE Computer Society, 1989, pp. 143-146.

(SHAW, 1990). M. Shaw, *Prospects for an Engineering Discipline of Software*, Technical Report CMU/SEI-90-TR-20, 1990.

(SHLAER e MELLOR, 1992). S. Shlaer, S. Mellor, *Object LifeCycles: Modeling the World in States*, 1 ed., Englewood Cliffs, N.J, Prentice-Hall, 1992.

(SHULL et al, 1996). F. Shull, W. Melo, V.Basili, *An Inductive Method for Discovering Design Patterns from Object Oriented Software Systems*, Technical Report UMIACS-TR-96-10, Universidade de Maryland, 1996.

(SZYPERSKI, 1998). C. Szyperski, *Component Software Beyond Object Oriented Programming*, 1 ed., Essex, Inglaterra, Addison-Wesley, 1998.

(TE-WEI e MARTIN, 1997). S. Te-Wei, S. Martin, "DEPARS: Design Pattern Recognition System". Em *VIII Conferência Internacional de Tecnologia de Software*, Curitiba, Junho, 1997.

(VASCONCELOS e WERNER, 97). F. M. Vasconcelos, C.M.L. Werner, *Uma Introdução aos Padrões de Software*, Relatório Técnico ES-458/97, COPPE-UFRJ, Fevereiro, 1997.

(VERSANT, 1997). Versant Corporation. *Argos*. Internet: http://www.versant.com.

(VLISSIDES et al., 1996). J. Vlissides, J. Coplien, N. Kerth. *Pattern Languages of Program Design 2*, 1 ed., Addison Wesley, 1996.

(VLISSIDES, 1997). J. Vlissides, "Patterns: The Top Ten Misconceptions". Em *Object Magazine*, SIGS Publications, Março, 1997.

(WERNER et al., 1999). C. Werner, M.Mattoso, R.Braga, M.Barros, L.Murta, A.Dantas. "Odyssey: Infra-estrutura de Reutilização baseada em modelos de domínio". Em *XIII Simpósio Brasileiro de Engenharia de Software*, Caderno de Ferramentas, Florianópolis, Outubro 1999 (aceito para publicação).

(WILDE e HUITT, 1992). N. Wilde, R. Huitt, "Maintenance Support for Object Oriented Programs". Em *IEEE Transactions on Software Engineering*, vol. 18, pp. 1038-1044, Dezembro, 1992.

(WINSEN, 1996). P. van Winsen, *Reengineering with Object Oriented Design Patterns*, Tese de Mestrado, Universidade de Utrecht, Novembro 1996.

(WIRFS-BROCK et al, 1990). R. E. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object Oriented Software*, 1 ed., Englewood Cliffs, N.J., Prentice-Hall, 1990.

(ZIMMER, 1994). W. Zimmer, "Relationships between Design Patterns". Em *Pattern Languages of Program Design (PloP"94)*, Reading, Massachusetts, Addison-Wesley, 1995.

(ZIMMER, 1998). W. Zimmer. "Experiences using Design Patterns to Reorganize an Object Oriented Application". Em *Proc. of the Workshop on OO Software Evolution and Re-Engineering*, ECOOP, Bruxelas, Bélgica, Julho, 1998.

Apêndice I

METAMODELO – Representação em Prolog

Este apêndice descreve os predicados que representam as construções básicas de um projeto orientado a objetos. Estes predicados foram definidos a partir do metamodelo para representação de construções de projeto orientado a objetos, descrito no capítulo 3.

Os principais elementos de projeto (pacotes, classificadores, operações, atributos, entre outros) são identificados pelo seu nome qualificado, isto é, o nome do elemento é precedido de todos os nomes que definem o escopo deste elemento. A Figura 75 ilustra o conceito de nome qualificado.

Classe Aluno definida no pacote BusinessClasses:

(nome qualificado: BusinessClasses::Aluno)

Atributo dataNascimento da Classe Aluno:

(nome qualificado: BusinessClasses::Aluno::dataNascimento)

Figura 75: Exemplos de nomes qualificados

I.1. Pacote

Um pacote é um mecanismo geral para organizar os elementos de um modelo em grupos. Pacotes bem projetados resultam do agrupamento de elementos semanticamente próximos, e que apresentem uma tendência de mudarem juntos. Desta forma, pacotes tendem a apresentar um fraco acoplamento entre si, uma forte coesão e um acesso bastante controlado aos elementos que o compõem.

A existência de um pacote em um projeto orientado a objetos é representado pelo

predicado *Pacote (NomePacote, Estereótipo, NomePacotePai),* onde:

- *<NomePacote>*: nome qualificado que identifica o pacote.
- *<Estereótipo>*: é uma forma de classificação dos pacotes prevista na UML (Exemplo: DatabaseAccess, UserInterface, BusinessClasses ...).
- *<NomePacotePai>*: se o pacote estiver contido em outro, este atributo indica o pacote onde ele está contido. Caso contrário, este atributo é definido com o valor "Nil".

I.2. Dependência entre Pacotes

Ao estabelecer uma relação de dependência entre dois pacotes A e B, onde, por exemplo, A dependa de B, os elementos visíveis de B passam a estar disponíveis para A. Através da organização dos elementos de um projeto em pacotes, e do controle de acesso a estes elementos através de relações de dependência entre os pacotes, o projetista pode controlar a complexidade de um projeto com um número grande de abstrações. A existência de uma relação de dependência entre pacotes é representada pelo predicado

DependenciaPacote (NomePacoteCliente, NomePacoteFornecedor), onde:

- *<NomePacoteCliente>*: nome qualificado do pacote dependente.
- *<NomePacoteFornecedor>*: nome qualificado do pacote cujos elementos visíveis passam a ficar disponíveis para o pacote cliente.

I.3. Classificador

Um classificador corresponde à definição de uma classe, interface ou tipo básico de dados. Uma Classe é a descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, métodos, relacionamentos e semântica. Uma classe pode representar um conceito do domínio do problema que está sendo modelado, como

também pode corresponder a elementos específicos do mundo computacional (universo de solução). Uma interface é a declaração de uma coleção de operações que, juntas, definem um serviço oferecido por um classificador que realize esta interface. Interfaces não podem possuir atributos, associações ou implementações de operações. Uma classe pode realizar uma ou mais interfaces, devendo fornecer uma implementação para cada operação definida em cada uma das interfaces realizadas. Um tipo básico de dados descreve um conjunto de valores sem identidade. Tipos básicos de dados correspondem normalmente a números, cadeias de caracteres, data e hora. Tipos básicos definidos pelo usuário são definidos como enumerações. Cada classificador presente em um projeto orientado a objetos é representado pelo predicado

Classificador (NomePacote, NomeClassificador, Estereótipo, Tipo, Abstrata, Folha, Raiz), onde:

- *<NomePacote>*: pacote onde o classificador foi definido.
- *<NomeClassificador>:* nome qualificado que identifica o classificador.
- *<Estereótipo>:* é uma classificação dada para a classe. Exemplo: O processo de desenvolvimento Rational Unified Process (JACOBSON et al., 1999) define três estereótipos básicos para as classes: *"Limitrofe", "Controle", "Entidade".*
- *<Tipo>*: define se o classificador corresponde a uma classe, uma interface ou um tipo básico de dados. Valores possíveis: *"Classe", "Interface"* e *"Data Type".*
- *<Abstrata>*: define se o classificador corresponde a um elemento instanciável ou não. Valores possíveis: *"Abstrata"* e *"Concreta"*. Para classificadores do tipo Interface, esta propriedade terá sempre o valor *"Abstrata"*.
- *<Folha>*: define se o classificador pode possuir elementos descendentes. Valores possíveis: *"Folha"* e *"NaoFolha".*
- *<Raiz>*: define se o classificador pode possuir elementos ascendentes, ou seja, se ele é a raiz de uma hierarquia de classificadores. Valores possíveis: *"Raiz"* e *"NaoRaiz".*

I.4. Visibilidade do Classificador no Pacote

Os classificadores definidos em um pacote podem ter diferentes visibilidades em relação a outros pacotes. Alguns classificadores podem ser visíveis fora deste pacote, enquanto que outros são visíveis apenas dentro do pacote onde eles foram definidos, ou seja, são utilizados internamente no pacote para auxiliar na implementação da funcionalidade fornecida pelos elementos visíveis.

O projetista pode controlar, através desta propriedade, a visibilidade entre os elementos de um sistema. Cada classificador pode ser definido como visível fora do pacote, ou visível apenas dentro do pacote. Esta propriedade é capturada pelo predicado *VisibilidadePacote (NomePacote, NomeClassificador, Visibilidade)*, onde:

- <*NomePacote*>: nome do pacote que contém o classificador.
- <*NomeClassificador*>: nome do classificador cuja visibilidade está sendo definida.
- <*Visibilidade*>: indica se o classificador é visível ou não fora do pacote. Valores possíveis: *"public"* (visível fora do pacote) e *"private"* (visível apenas dentro do pacote).

I.5. Realização de uma Interface

A realização de uma interface representa o relacionamento entre uma especificação e sua implementação. Uma especificação descreve o comportamento ou estrutura de algo sem determinar sua implementação. A semântica deste relacionamento indica que o realizador (uma classe) deve definir uma implementação para cada operação especificada pelo realizado (normalmente uma interface). A presença de um relacionamento de realização entre classes e interfaces é representada pelo predicado

Realiza (NomeClassificadorRealizador, NomeClassificadorRealizado), onde:

- <*NomeClassificadorRealizador*>: corresponde ao classificador que fornece a implementação (Classe).
- <*NomeClassificadorRealizado*>: corresponde ao classificador que define a

especificação a ser realizada (Interface).

I.6. Herança

Herança é um mecanismo através do qual elementos mais específicos podem incorporar a estrutura e o comportamento definido por elementos mais gerais. Este relacionamento é capturado pelo predicado

HerdaDe (NomeClassificadorEspecífico, NomeClassificadorGenérico), onde:

- <*NomeClassificadorEspecífico*>: corresponde ao elemento derivado.
- <*NomeClassificadorGenérico*>: corresponde ao elemento base.

I.7. Atributo

Representa um atributo simples ou uma associação com um classificador (pseudo-atributo conforme definido na UML). Um atributo simples é um elemento de uma classe que descreve um conjunto de valores que instâncias desta classe devem manter. Normalmente, um atributo simples é definido para valores sem identidade (tipos básicos de dados), sendo semanticamente equivalente a uma associação de composição. Neste metamodelo, o predicado Atributo é utilizado tanto para capturar a existências destes atributos simples como também dos pseudo-atributos. Pela definição da UML, um pseudo-atributo representa o papel desempenhado por um classificador em um relacionamento. A UML define três tipos básicos de relacionamentos: agregação, composição e associação. Agregação define um relacionamento todo-parte entre um agregado e suas partes constituintes. Composição é uma forma mais forte de agregação, onde um elemento pode ser parte de apenas um todo e além disso, o todo tem responsabilidade de criação e destruição das suas partes. Associação é um relacionamento que envolve a conexão entre instâncias das classes associadas sem a existência de uma semântica do tipo todo-partes. Quando o relacionamento entre dois classificadores A e B for navegável de A para B, isto implica na existência de um pseudo-atributo em A com o nome do papel exercido por B no relacionamento. O

mesmo ocorre, de forma análoga, quando o relacionamento for navegável de B para A.

Os atributos simples definidos diretamente na classe, bem como as associações entre os classificadores, são capturados pelo seguinte predicado

Atributo (NomeClassificador, NomeAtributo, Escopo, Visibilidade, TipoAtributo, ClassificadorAssociado, Modificabilidade, Multiplicidade, Agregação), onde:

- <*NomeClassificador*>: define o classificador onde o atributo foi definido.
- <*NomeAtributo*>: nome do atributo ou do papel (pseudo-atributo) correspondente ao classificador associado.
- <*Escopo*>: define se o atributo possui apenas um valor para a classe, ou seja, todas as instâncias compartilham o mesmo valor (*"Classe"*), ou se cada instância tem o seu valor independente (*"Instancia"*).
- <*Visibilidade*>: define a visibilidade do atributo em relação a outros classes. Valores válidos: *"Public"* (visível por outros classes), *"Protected"* (visível apenas para as subclasses), *"Private"* (visível apenas dentro da classe).
- <*TipoAtributo*>: define o tipo do atributo, de acordo com a seguinte regra: se o atributo tiver multiplicidade igual a 1, este atributo indica o classificador que define as características do atributo. Se o atributo tiver multiplicidade diferente de 1 (multivalorado), este atributo indica o classificador utilizado para definir a classe coleção que armazenará os diversos valores (exemplo: *vector, list, set, ...*).
- <*ClassificadorAssociado*>: define o tipo do atributo, independente da multiplicidade. Para atributos de multiplicidade 1, *TipoAtributo* e *ClassificadorAssociado* são equivalentes. Caso contrário, *TipoAtributo* indica a classe coleção e *ClassificadorAssociado* indica o tipo de cada elemento da coleção. Exemplo: um vetor de alunos (*TipoAtributo:* "vector" e *Classificador Associado:* "Aluno").
- <*Modificabilidade*>: indica se o atributo corresponde a um valor constante ou não. Valores possíveis: *"Const"*, *"NonConst"*.
- <*Multiplicidade*>: indica se o atributo é multivalorado ou não. Valores possíveis:

"1" e "n".

- *<Agregação>*: indica o tipo de relacionamento existente entre o classificador que possui o atributo e o classificador que define o tipo do atributo. Valores possíveis: *"Composição", "Agregação"* e *"Associação"*.

I.8. Operação

Uma operação é a especificação de um serviço que pode ser requisitado a um objeto. Uma operação possui uma assinatura que descreve os parâmetros possíveis na sua invocação, incluindo valores de retorno. Um método é a implementação de uma operação, especificando o algoritmo ou procedimento que produz os resultados da operação. Neste trabalho, os conceitos de operação e método são definidos pelo mesmo predicado. A diferenciação entre os dois conceitos é feita por uma propriedade, indicando se o elemento corresponde apenas a uma definição (operação) ou se corresponde a uma implementação (método). Cada operação ou método definido pelos classificadores é representado pelo predicado

Operacao (NomeClassificador, NomeOperação, Escopo, Visibilidade, Estereótipo, Pol*imórfica, Abstrata, Const)*, onde:

- *<NomeClassificador>*: indica o classificador onde a operação foi definida.
- *<NomeOperação>*: define o nome da operação.
- *<Escopo>*: define se a operação se aplica a uma instância individualmente (*"Instancia"*) ou à classe (*"Classe"*).
- *<Visibilidade>*: define a visibilidade externa desta operação (*"Public", "Protected", "Private"*).
- *<Estereótipo>*: classifica a operação em alguns tipos padronizados. (*"Constructor", "Destructor", "Read Accessor", "Write Accessor"*), onde *Constructor* corresponde a uma operação que resulta na instanciação de um novo objeto da classe, *Destructor* corresponde a uma operação de destruição de um objeto da classe, *Read Accessor*

corresponde ao retorno simples de um atributo pertencente à classe, e *Write Accessor* corresponde a uma simples atribuição efetuada em um atributo da classe. As operações de estereótipo *Read Accessor* e *Write Accessor* servem para esconder o formato de armazenamento dos atributos, permitindo o acesso aos mesmos apenas através destas operações. Outros estereótipos podem ser definidos além dos aqui citados.

- *<Polimórfica>*: define se a operação pode ser definida em subclasses. Valores possíveis: *"Final"* (não pode) e *"Virtual"* (pode).
- *<Abstrata>*: define se este elemento corresponde apenas a uma definição de uma operação, ou se corresponde à implementação de um método. Valores possíveis: *"Abstrata"* (definição) e *"Metodo"* (implementação).
- *<Const>*: define se a operação preserva o estado do objeto/classe ou não. Valores possíveis: *"Const"* (preserva) e *"NonConst"* (não preserva).

I.9. Parâmetro

Corresponde à especificação de um elemento que pode ser passado, modificado ou retornado por uma operação. Um parâmetro possui um nome, um tipo e uma direção. Os parâmetros de cada operação capturada no modelo são representados pelo predicado

Parametro (NomeOperação, NomeParâmetro, Ordem, Direção, TipoParâmetro), onde:

- *<NomeOperação>*: define a operação onde este parâmetro é esperado.
- *<NomeParâmetro>*: define o nome do parâmetro.
- *<Ordem>*: define a ordem do parâmetro dentro do conjunto de parâmetros da operação (primeiro parâmetro corresponde ao valor "1").
- *<Direção>*: define a direção do parâmetro, conforme definição da UML. Valores possíveis: *"In"* – entrada por valor, *"Out"* – resultado disponível para o chamador após a execução da operação, *"InOut"* – entrada que pode ser modificada e *"Return"* – retorno de função. Não existe diferrência semântica em relação a um

parâmetro *Out*, apenas este valor está disponível para utilização em uma expressão).

- *<TipoParâmetro>*: indica o classificador que define as características do parâmetro.

I.10. Criação de um objeto

Objetos são criados em tempo de execução como resultado da invocação de operações específicas de criação de objetos. Estas operações são conhecidas como construtores, e a sua invocação indica uma dependência de uso entre a operação que invoca esta criação e a classe que será instanciada. Toda invocação de um construtor é capturada pelo predicado

Instancia(NomeOperaçãoChamadora,NomeClassificadorInstanciado, NomeOperaçãoConstrução), onde:

- *<NomeOperaçãoChamadora>*: método que está invocando a construção.
- *<NomeClassificadorInstanciado>*: classe que está sendo instanciada.
- *<NomeOperaçãoConstrução>*: operação de construção que está sendo invocada.

I.11. Destruição de um objeto

Representa um momento onde um objeto é destruído. A destruição de um objeto corresponde à invocação de uma operação conhecida pelo estereótipo de destrutor. Toda invocação de um destrutor é capturada pelo predicado

Destroi(NomeOperaçãoChamadora, NomeClassificadorDestruído, NomeOperaçãoDestruição), onde:

- *<NomeOperaçãoChamadora>*: método que está invocando a destruição.
- *<NomeClassificadorDestruído>*: classificador que define a operação de destruição que está sendo invocada. Como várias linguagens implementam destrutores virtuais, este atributo pode não corresponder exatamente à classe do objeto que será destruído. Ele indica apenas a visibilidade, ou dependência de tipo, existente entre a

operação que está solicitando a destruição e o classificador.

- *<NomeOperaçãoDestruição>*: operação de destruição que está sendo invocada. Note que devido a uma possível invocação polimórfica, o método exato que é chamado depende da execução do programa.

I.12. Chamada de uma operação

Uma chamada de operação é a invocação síncrona de um procedimento, onde a operação chamadora aguarda o término da operação invocada para voltar a ter o controle de execução. Cada chamada de operação é capturada pelo predicado:

Invoca(NomeOperaçãoChamadora, NomeClassificadorInvocado, NomeOperaçãoInvocada, ElementoAcessado), onde:

- *<NomeOperaçãoChamadora>*: método que está invocando a operação.

- *<NomeClassificadorInvocado>*: classe que define a operação que está sendo invocada. Em função do polimorfismo, este termo pode não corresponder exatamente à classe do objeto que será referenciado. Ele indica apenas qual a visibilidade, ou dependência de tipo, que existe entre o método solicitante e o classificador.

- *<NomeOperaçãoInvocada>*: operação que está sendo invocada.

- *<ElementoAcessado>*: Indica como o método obteve acesso ao elemento cuja operação está sendo solicitada. Valores válidos: *"Self"* (invocação de uma operação do próprio objeto), *"Atributo"* (invocação de uma operação de um elemento que é definido como um atributo ou pseudo-atributo do chamador), *"Parametro"* (o elemento foi recebido como parâmetro da operação chamadora), *"Global"* (o elemento está disponível globalmente para o sistema), *"Local"* (o elemento chamado foi criado localmente na operação chamadora), *"Local Indireto"* (o elemento chamado foi obtido através da invocação de uma operação de um outro objeto, normalmente uma chamada em cadeia, como por exemplo a invocação da operação obterPreco em: fita.obterFilme().obterPreco().

I.13. Acesso a um atributo

A implementação dos métodos de uma classe envolve, freqüentemente, a manipulação de atributos do objeto ou da classe envolvida. Embora não seja recomendado, é possível implementar em muitas linguagens de programação o acesso direto a atributos de outros objetos da mesma classe ou até mesmo de outras classes. O acesso a atributos da mesma classe ou de outras é representado pelo predicado:

Acesso (NomeOperação, AtributoAcessado), onde:

- *<NomeOperação>*: nome do método que está acessando o atributo.
- *<AtributoAcessado>*: nome qualificado do atributo que está sendo acessado.